

软件工程

第四章 软件架构设计

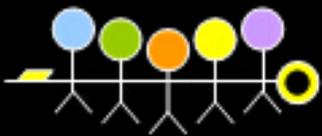
4-4 面向NFR的架构设计思想与案例





主要内容

- 1 软件架构设计的四大思想
- 2 用于描述软件架构的部署模型
- 3 软件架构设计的案例
 - Struts MVC → Spring MVC
 - 网站架构的演变
 - 大型多人在线游戏(MMO)
 - Facebook的开放API架构
 - 面向服务的移动应用架构
 - 微博的消息架构分析



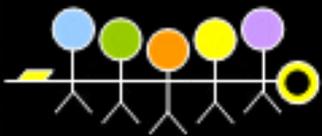
1 软件架构设计的四大思想



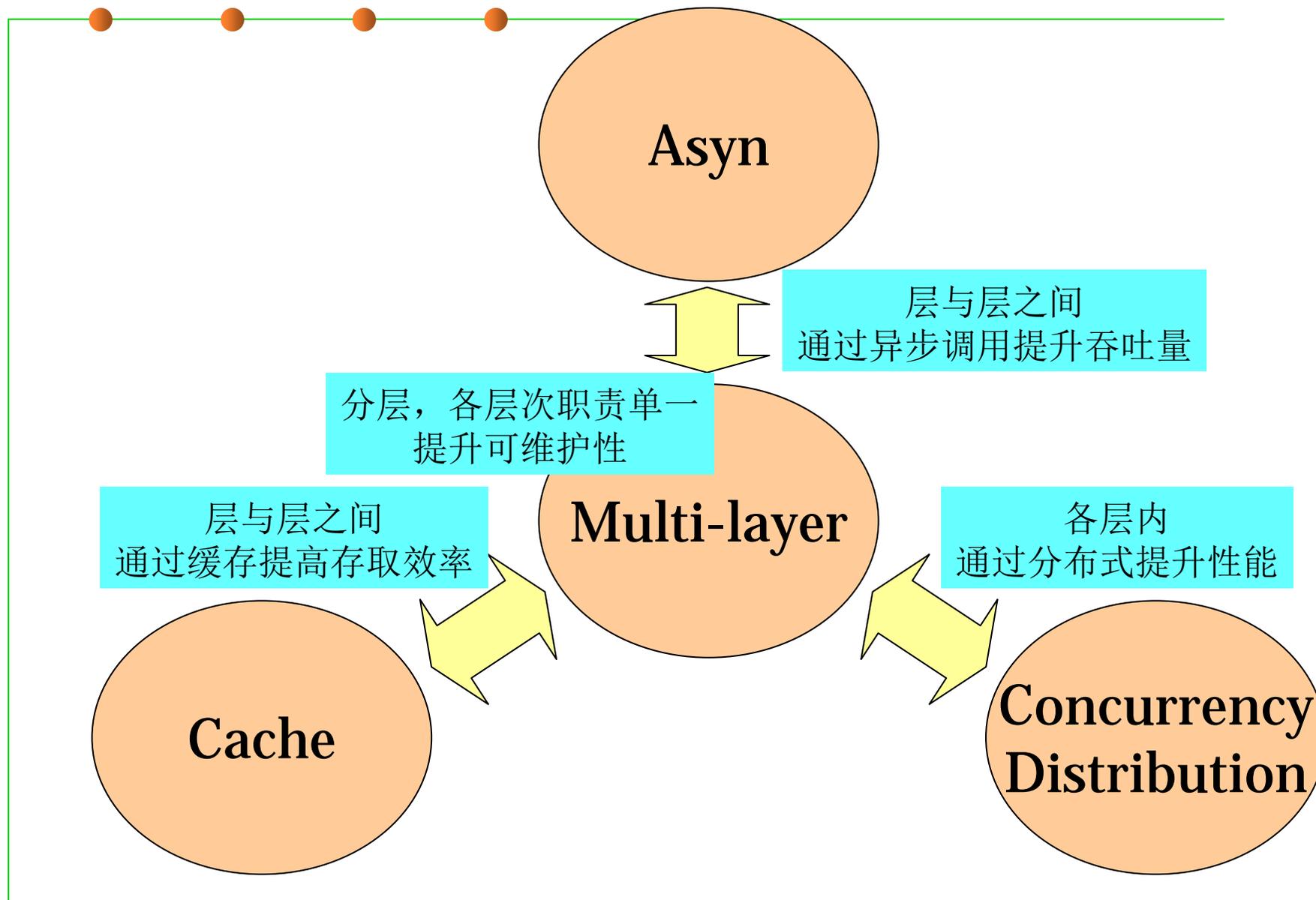


软件架构的基本模式

- **分层：C/S、B/S、多层，数据、计算与显示的分离(MVC)**
 - 一个模块做很多事情 → 各负其责，分工明确
 - 牺牲了效率，提升了可维护性。
- **异步：事件、消息**
 - 请求之后等待结果(同步) → 请求之后继续执行，后续等待结果(异步)
 - 性能(吞吐量)提高，但实时性变差；
- **缓存：页面缓存、数据缓存、消息缓存**
 - 直接到源头去取 → 预取
 - 提高了效率，但牺牲了准确性
- **并发(分布式)：集群、负载均衡、分布式数据库**
 - 原本一个模块处理很多请求 → 多个模块共同处理这些请求
 - 提高了吞吐量、响应时间、可靠性，但需要考虑同步等复杂问题

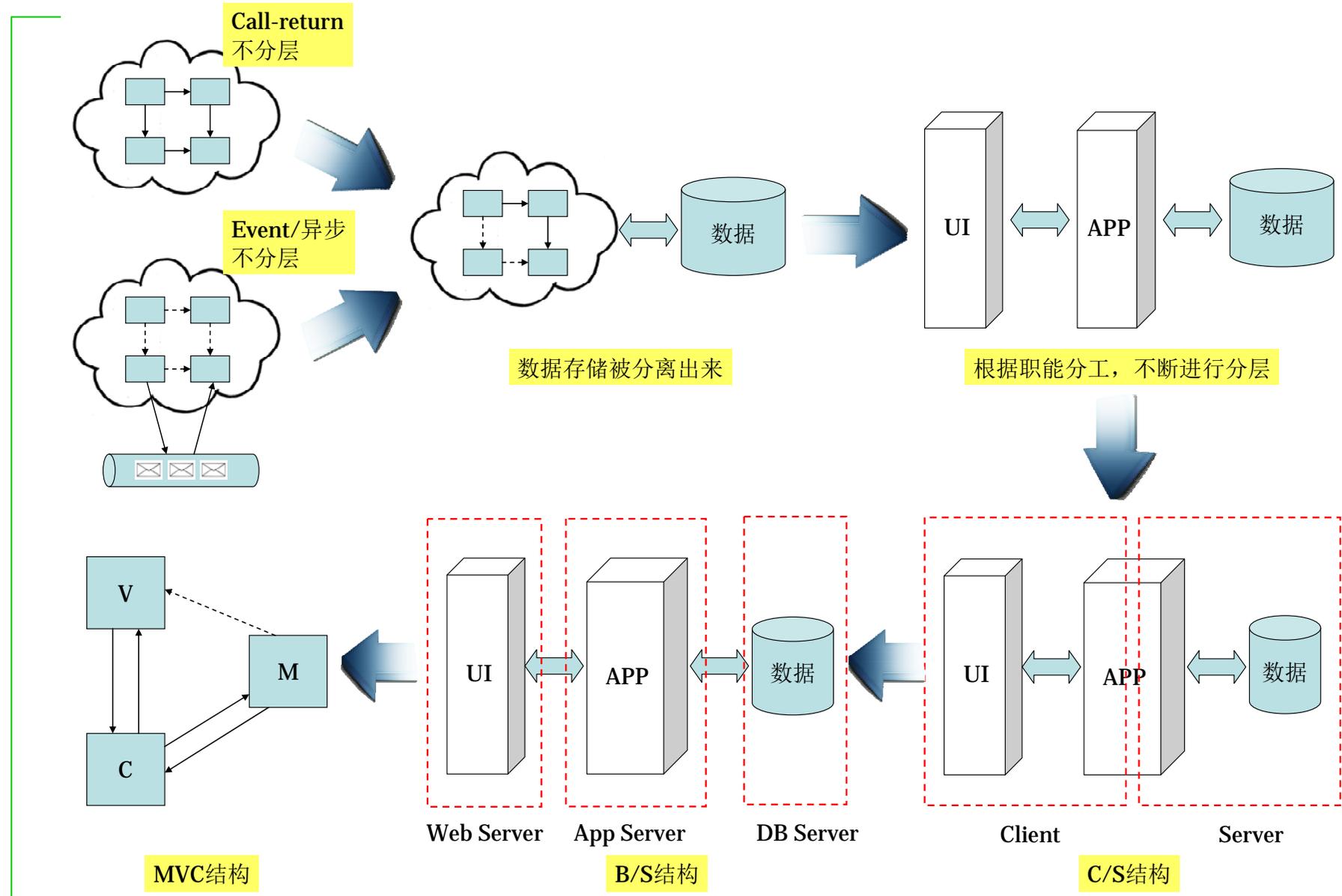


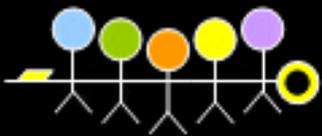
“软件架构四大器”





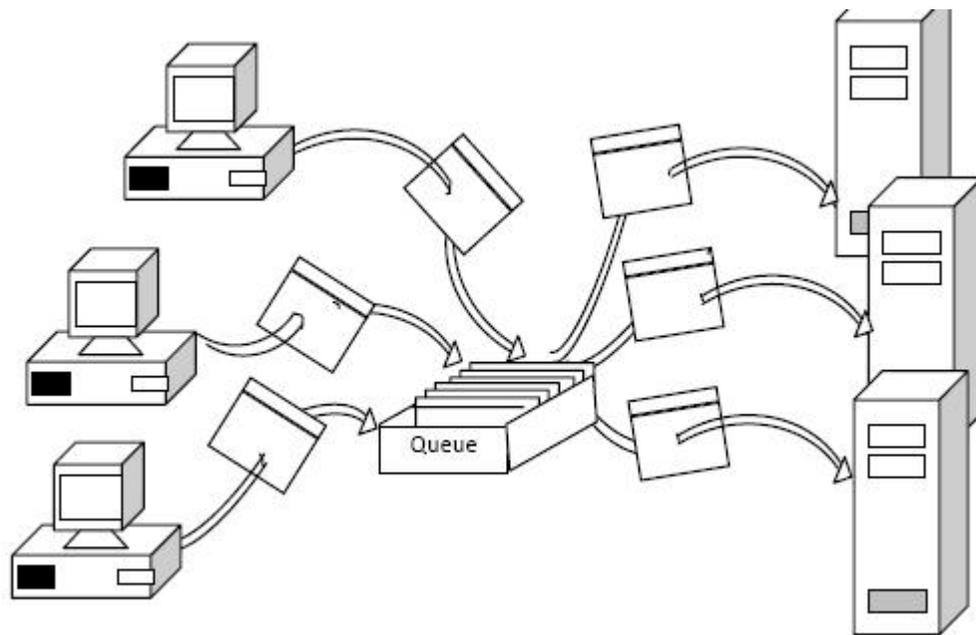
分层





异步

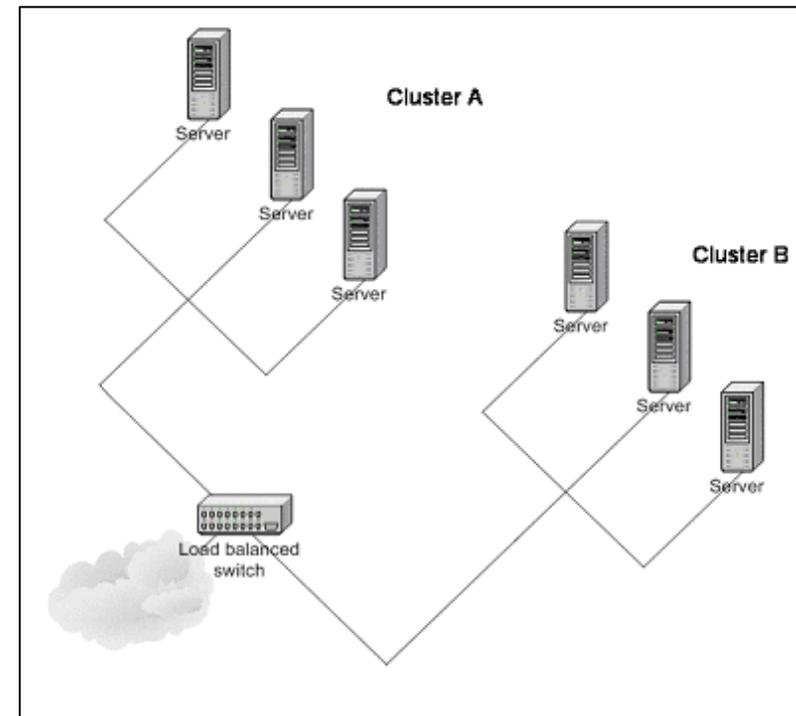
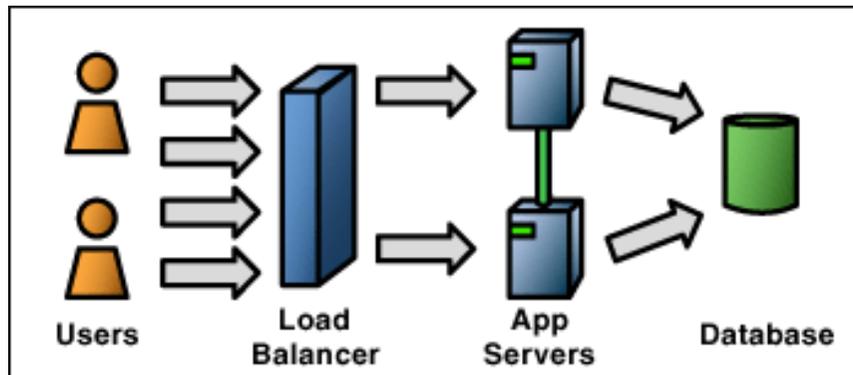
- 通过“第三者”——消息——完成模块之间的功能调用。





分布式+并发

- 事实上，功能层**并不一定只驻留在同一台服务器上**，数据层也是如此；
- 如果**功能层(或数据层)分布在多台服务器上**，那么就形成了**基于集群(Cluster)的C/S物理分布模式**。



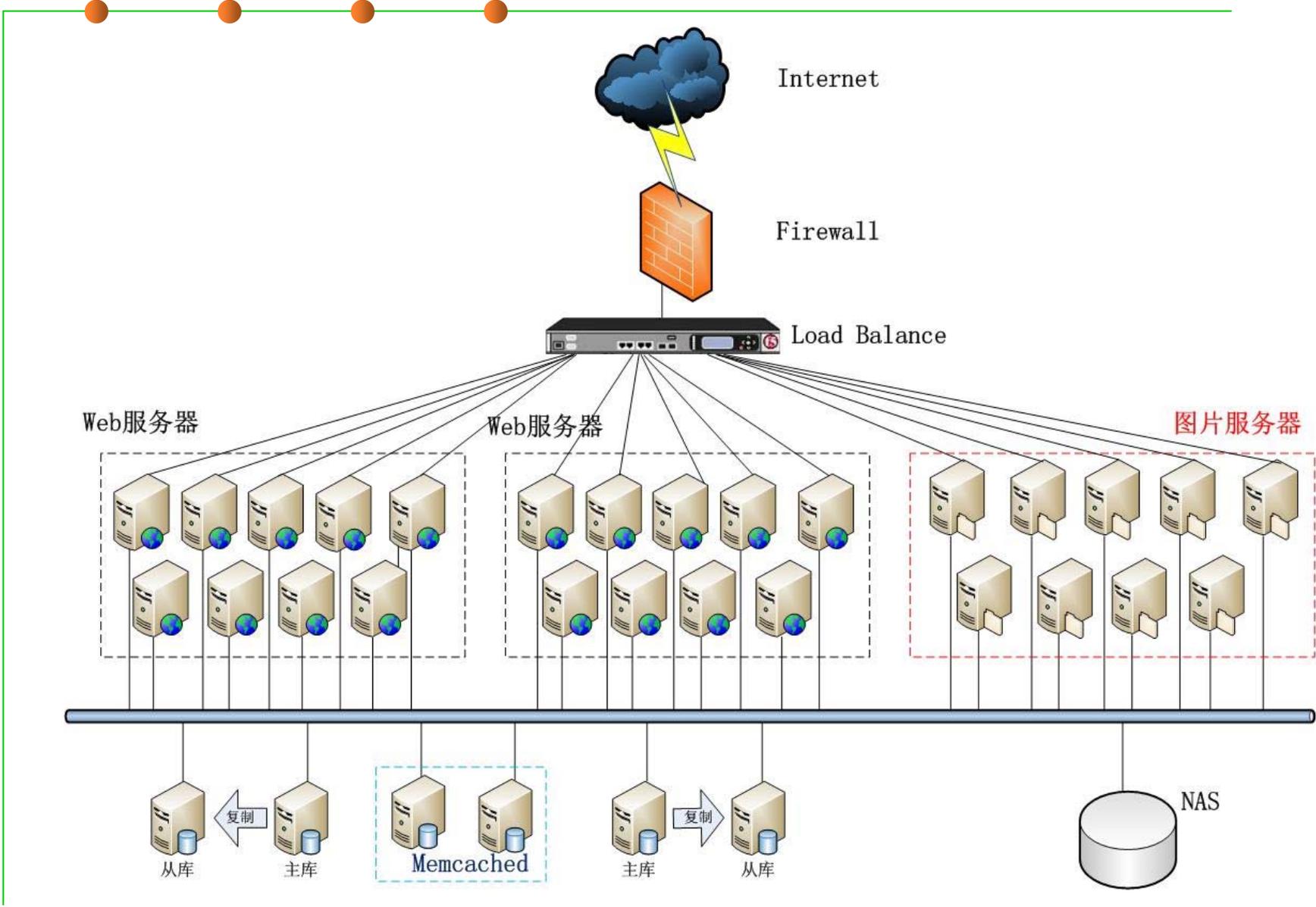


集群(Cluster)

- A cluster is a group of loosely coupled servers that work together closely so that in many respects it can be viewed as though it were a single server. (一组松散耦合的服务器，共同协作，可被看作是一台服务器)
- Clusters are usually deployed to improve **speed, reliability and availability** over that provided by a single server, while typically being much more **cost-effective** than single computers of comparable speed or reliability. (用来改善速度、提高可靠性与可用性，降低成本)
- Load-balancing is a key issue in clusters. (负载均衡是集群里的一个关键要素)
- Physical cluster and logical cluster(物理集群、逻辑集群)



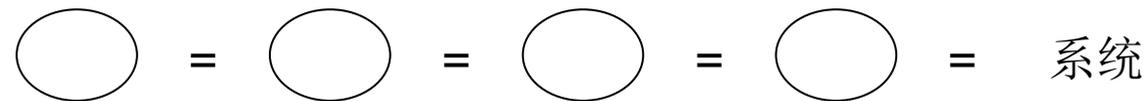
集群(Cluster)



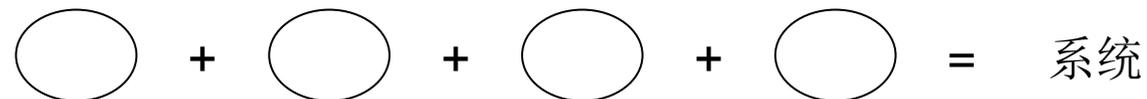


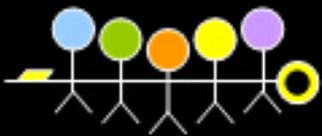
集群的方法

- 集群内各服务器上的内容保持一致(通过并发/冗余提高可靠性与可用性)



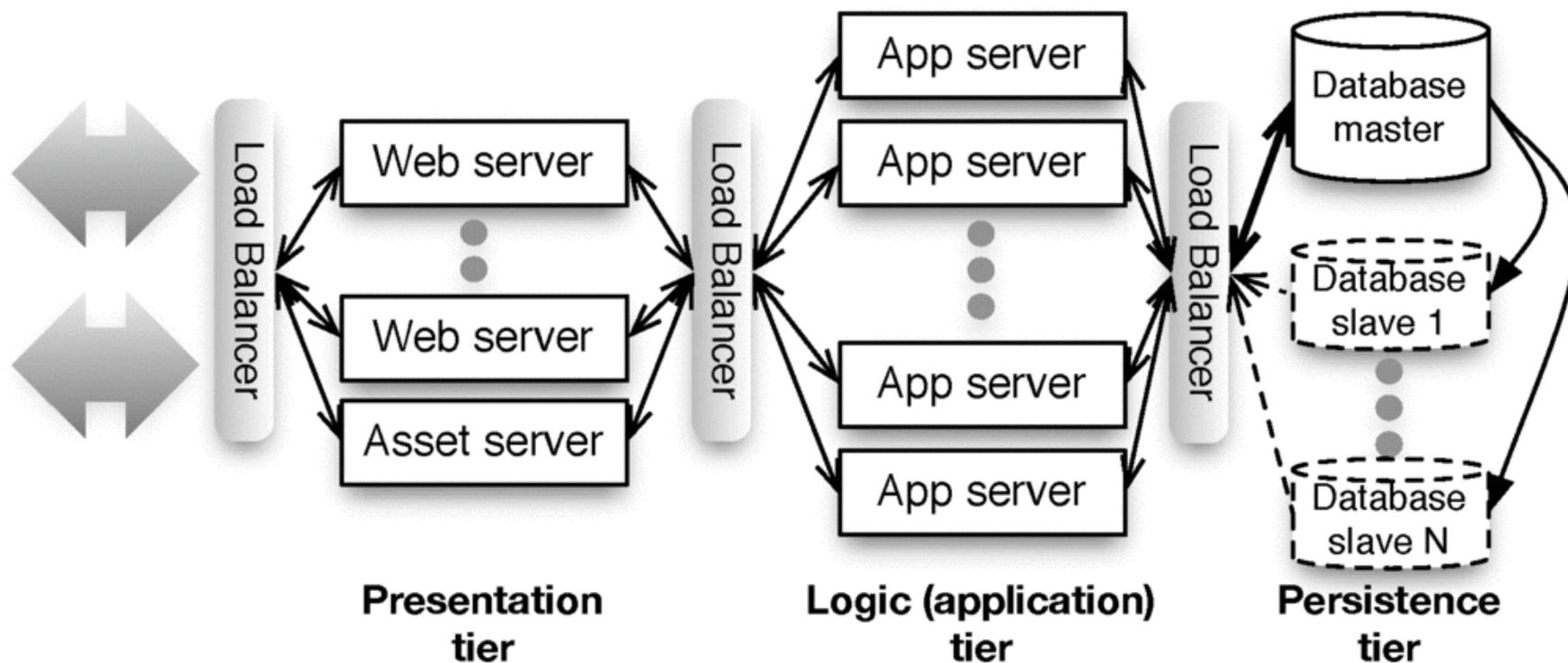
- 集群内各服务器上的内容之和构成系统完整的功能/数据，是对系统功能集合/数据集合的一个划分(通过分布式提高速度与并行性)

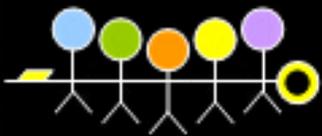




负载均衡(Load Balance)

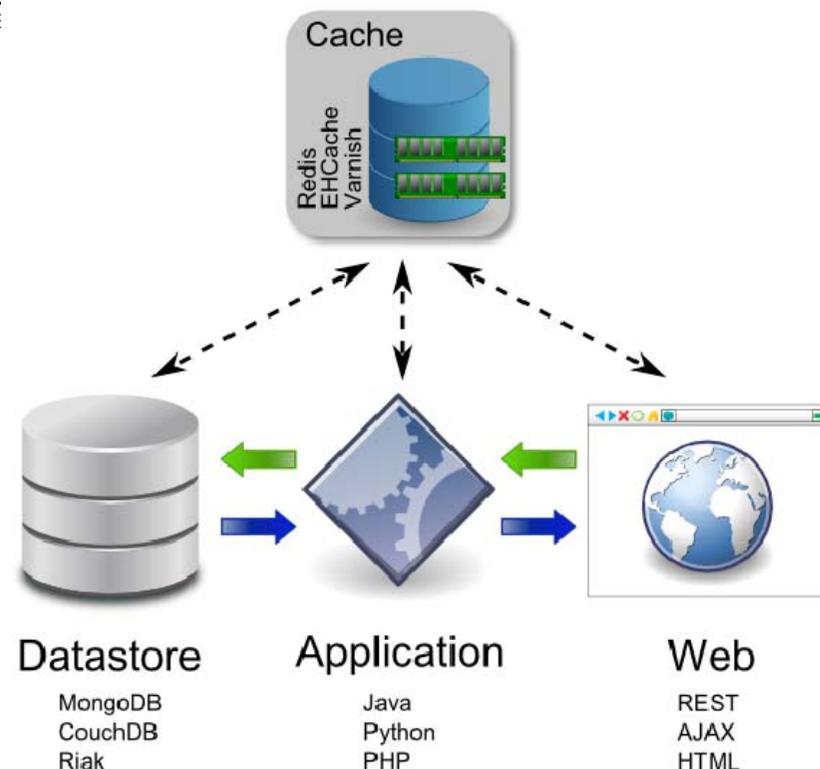
- 三个层次的负载均衡





缓存(cache)

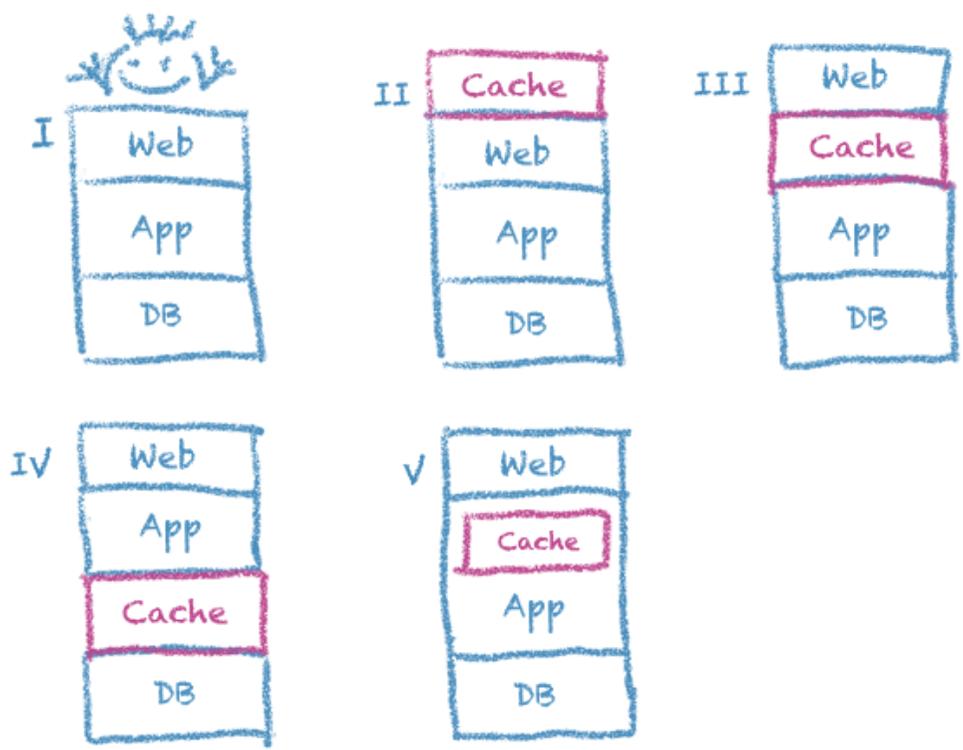
- 缓存：解决web应用性能瓶颈。
 - “缓存就像清凉油，哪里不舒服，抹一下就好了”
- 性能瓶颈的体现：高延时、拥塞和服务器负载
 - “下载HTML只需要总的用户响应时间的10-20%，剩下的80-90%全部用于下载页面中的其它组成内容(如各种图像等)”
- 缓存：
 - 一种临时存储，将数据复制到不同于原始数据源的位置，访问缓存数据的速度比访问原始数据的速度要快得多，从而可以减小服务器负载和带宽消耗，提高用户性能。

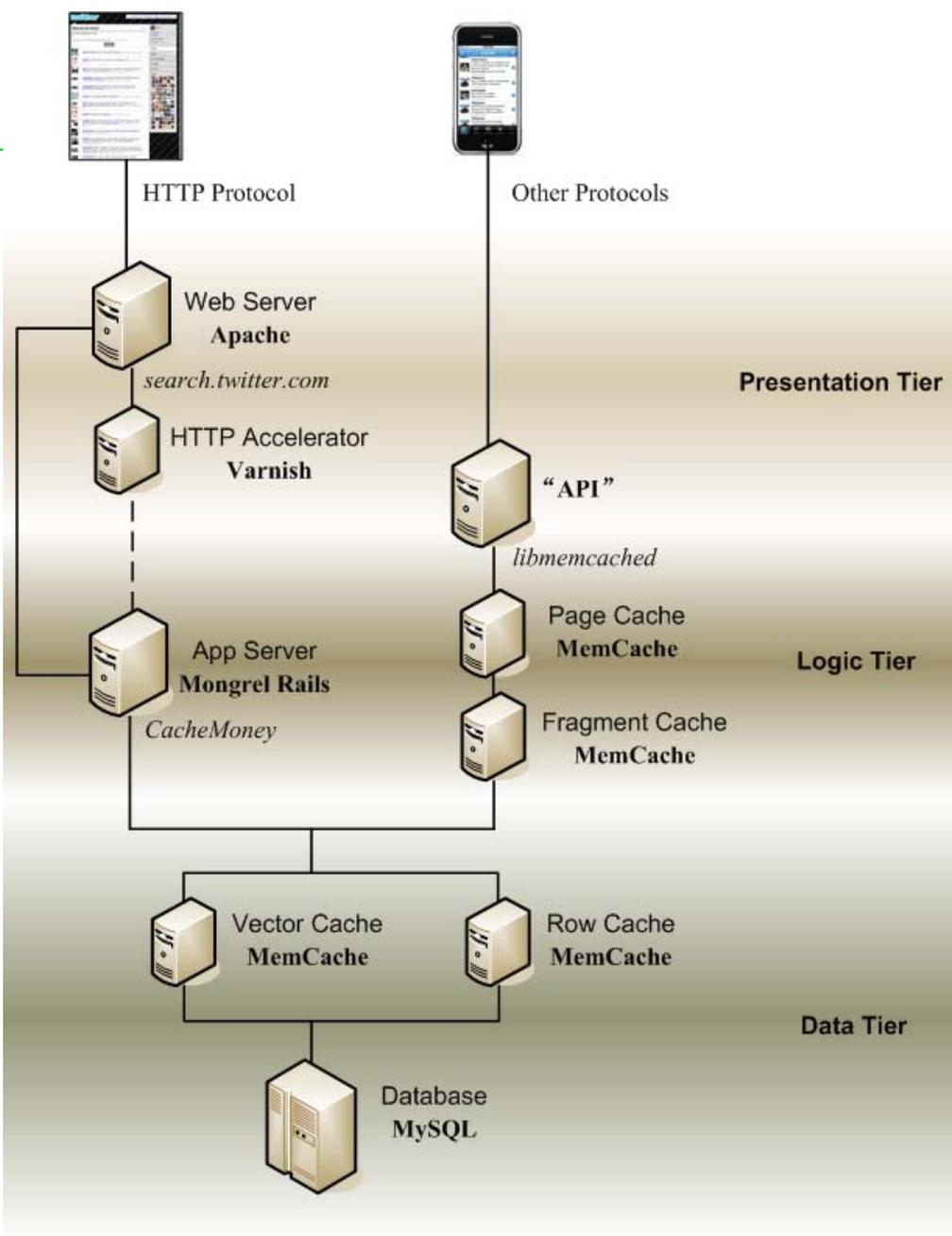
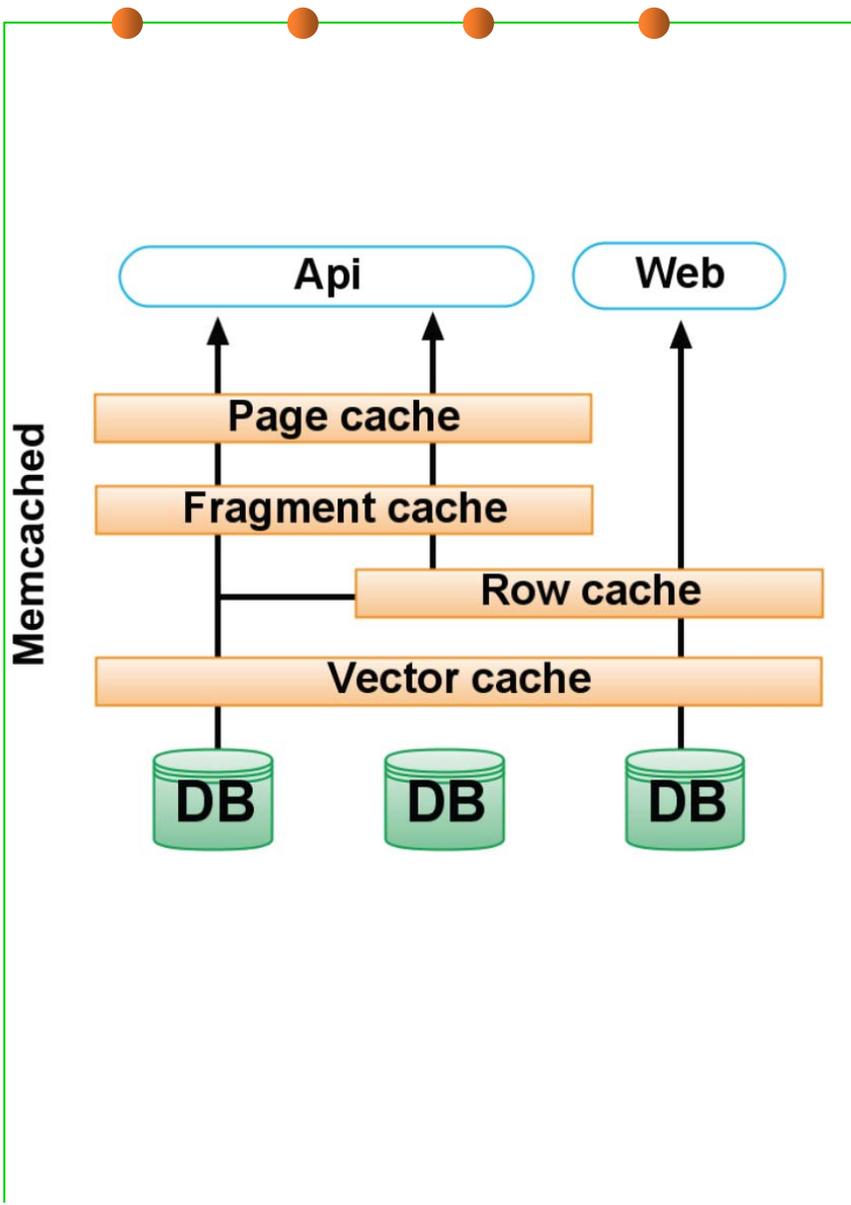




缓存(cache)

- Cache可以处在多个不同的位置

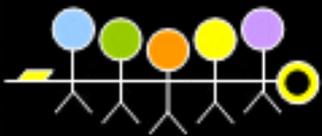






归纳：软件架构的基本模式

- **分层：C/S、B/S、多层，数据、计算与显示的分离(MVC)**
 - 一个模块做很多事情→各负其责，分工明确
 - 牺牲了效率，提升了可维护性。
- **异步：事件、消息**
 - 请求之后等待结果(同步)→请求之后继续执行，后续等待结果(异步)
 - 性能(吞吐量)提高，但实时性变差；
- **缓存：页面缓存、数据缓存、消息缓存**
 - 直接到源头去取→预取
 - 提高了效率，但牺牲了准确性
- **并发(分布式)：集群、负载均衡、分布式数据库**
 - 原本一个模块处理很多请求→多个模块共同处理这些请求
 - 提高了吞吐量、响应时间、可靠性，但需要考虑同步等复杂问题



2 用于描述软件架构的部署模型





软件部署模型 (Deployment Model)

- 反映了系统中软件和硬件的物理架构，表示系统运行时的处理节点以及节点中对象/子系统的分布与配置。
 - 部署对系统的性能和复杂度具有较大影响，需要在设计初期就要完成。
- 描述系统中各硬件之间的物理通讯关系；
- 描述各软件实体被配置到哪个具体硬件上、这些软件实体之间的物理通讯关系；



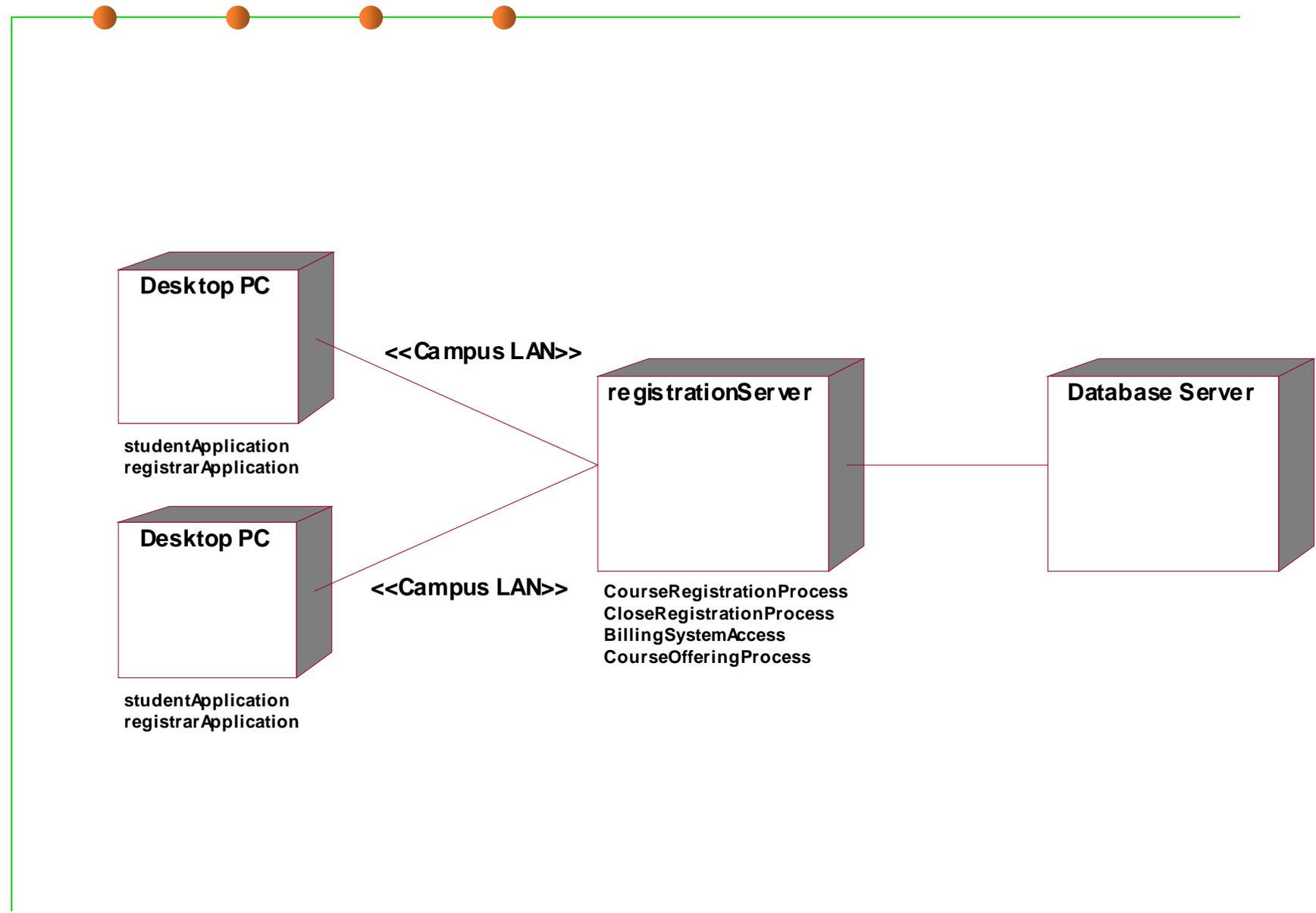
软件部署模型 (Deployment Model)

- 为系统选择硬件配置和运行平台；
- 将类、包、子系统分配到各个硬件节点上。

- 系统通常使用分布式的多台硬件设备，通过UML的部署图 (deployment diagram) 来描述；
 - 部署图反映了系统中软件和硬件的物理架构，表示系统运行时的处理节点以及节点中对象/子系统的分布与配置。
 - 部署对系统的性能和复杂度具有较大影响，需要在设计初期就要完成。



部署子系统



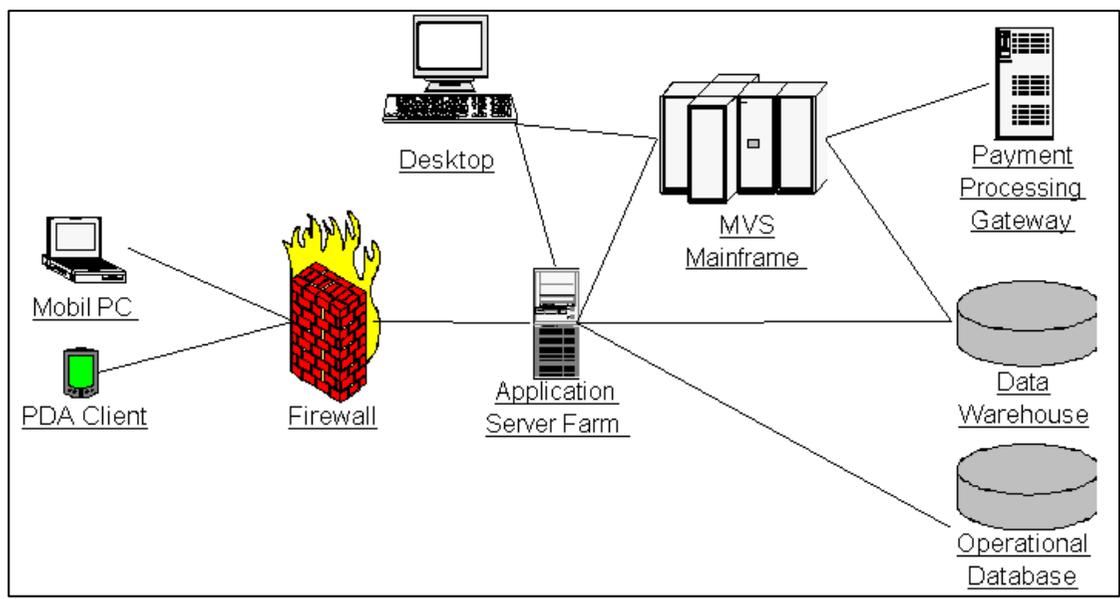
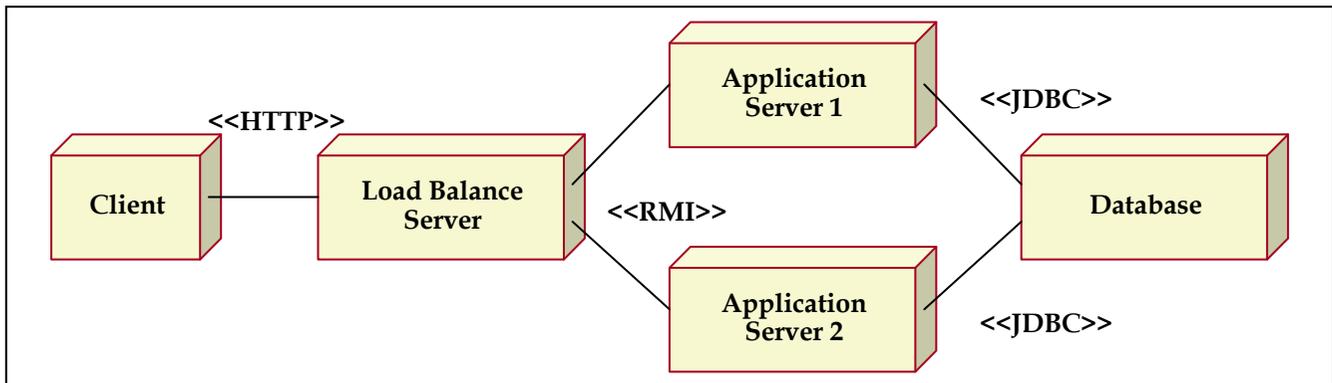


关于部署图

- **部署图(deployment diagram):**
 - 节点(node): 一组运行资源, 如计算机、设备或存储器等。每个节点用一个立方体来表示,
 - 节点的命名: client、Application Server、Database Server、Mainframe等较通用的名字;
 - 节点立方体之间的连接表示这些节点之间的通信关系, 通常有以下类型: 异步、同步; HTTP、SOAP; JDBC、ODBC; RMI、RPC; 等等;
- **部署图在两个层面的作用:**
 - High-level: 描述系统中各硬件之间的物理通讯关系;
 - Low-level: 描述各软件实体被配置到哪个具体硬件上、这些软件实体之间的物理通讯关系;

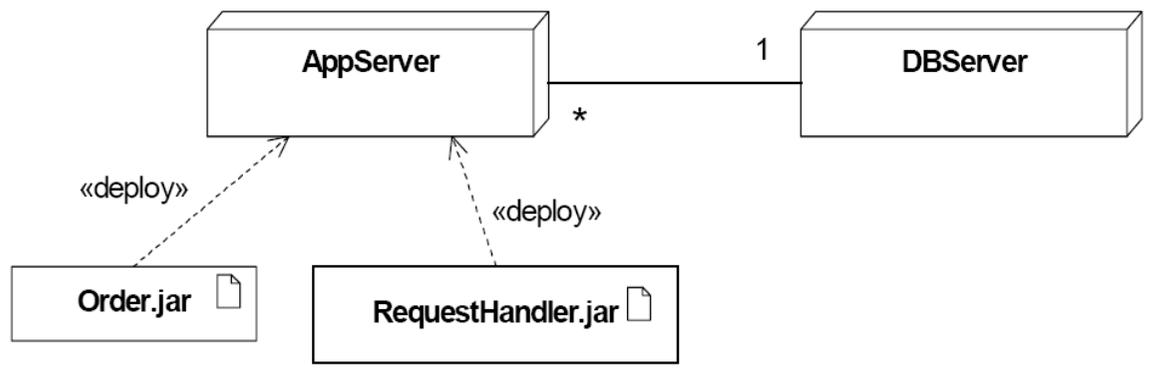
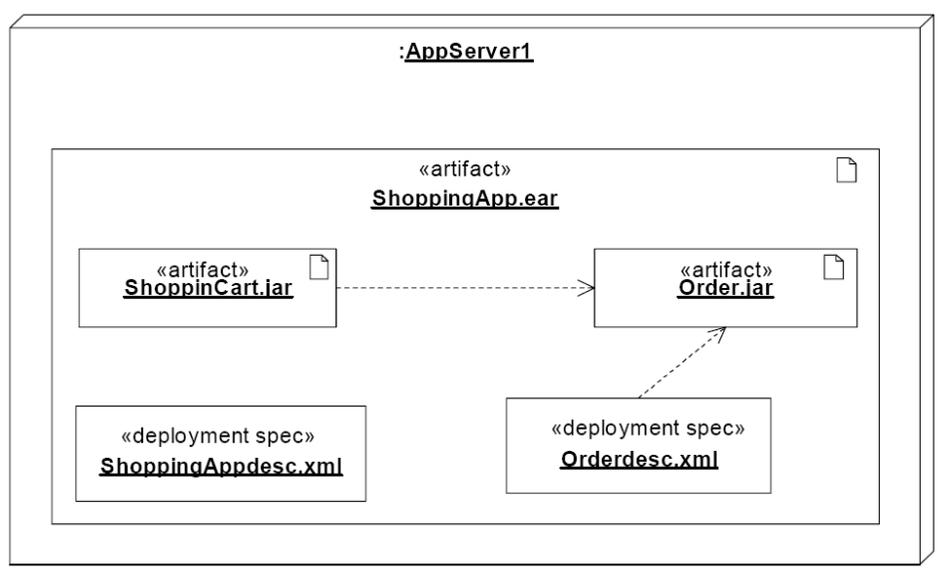


High-level Deployment Diagram





Low-level Deployment Diagram





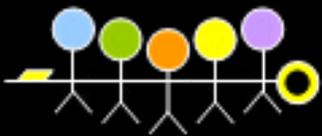
绘制部署图(deployment diagram)

- **确定“节点(node)”：**
 - 标示系统中的硬件设备，包括大型主机、服务器、前端机、网络设备、输入/输出设备等。
 - 一个处理机(processor)是一个节点，它具有处理功能，能够执行一个组件；
 - 一个设备(device)也是一个节点，它没有处理功能，但它是系统和外部世界的接口。
- **对节点加上必要的“构造型(stereotype)”**
 - 使用UML的标准构造型或自定义新的构造型，说明节点的性质。
- **确定“连接(connection)”**
 - 把系统的包、类、子系统、数据库等内容分配到节点上，并确定节点与节点之间、节点与内容之间、内容与内容之间的联系，以及它们的性质。
- **绘制配置图(deployment diagram)**



架构设计思路小结

- 逻辑架构：只考虑如何分层、每个层次中的模块、层次内模块的关系、层次间模块的关系。主要是给开发者提供指南。
- 物理架构：考虑的是实际硬件/网络环境，以及如何将逻辑架构映射到硬件/网络环境上去。主要是给实施人员提供指南。
 - 通常，逻辑分层可以1:1映射到物理分层上；
 - 某些时候，多个逻辑层次可以部署在同一个物理层次上(n:1)；
 - 某些时候，同一个逻辑层次可以拆分为多个物理设施上(1:n)；
- 物理架构的设计思路：
 - 从逻辑架构入手，分别考虑每个逻辑层次在物理环境下是如何实现的；
 - 从简单到复杂，考虑每项设计决策对物理设施的要求，逐渐扩充物理架构。



3 软件架构设计案例分析





3.1 Struts MVC → Spring MVC





Spring: J2EE + AOP + IOC + DAO + ORM +MVC

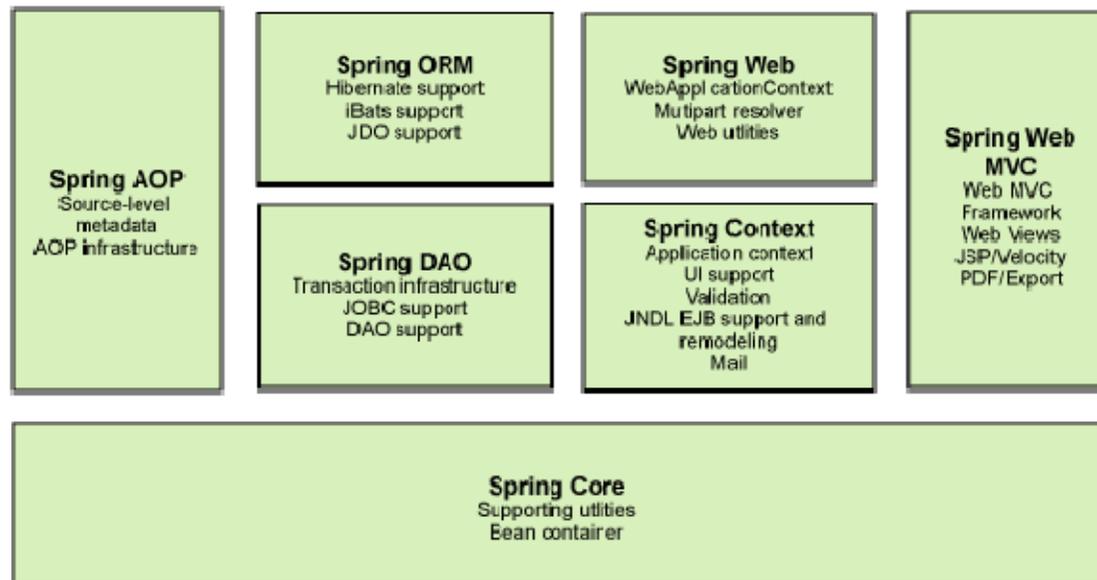
- Spring: 基于MVC的开源J2EE框架
- <http://www.springframework.org/>



- 三大优势特征:
 - 依赖注入、控制反转(Dependency Injection, Inversion of Control)
 - 面向方面的编程(Asspect-Oriented Programming, AOP)
 - 与J2EE第三方开放标准的集成(例如Struts, Hibernate等)



Spring: J2EE + AOP + IOC + DAO + ORM + MVC





Spring框架的七大构件(1)

■ 核心容器:

- 核心容器提供Spring框架的基本功能。核心容器的主要组件是BeanFactory，它是工厂模式的实现，使用控制反转(Inverse of Control, IOC)模式将框架API与用户开发的JAVA代码分开，消除二者之间的依赖，使开发变得简单；

■ Spring上下文:

- Spring上下文是一个配置文件，向Spring框架提供上下文信息。Spring上下文包括企业服务，例如JNDI、EJB、电子邮件、国际化、校验和调度功能。

■ Spring AOP:

- 通过配置管理特性，Spring AOP (Aspect Oriented Programming)模块直接将面向方面的编程功能集成到了Spring框架中，通过将影响多个类的行为封装到可复用的模块中，进而以“声明”而非“编码”的方式将aspect与JAVA class松散绑定到一起。



Spring框架的七大构件(2)

■ Spring DAO:

- JDBC DAO抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。

■ Spring ORM:

- Spring框架插入了若干个ORM框架，从而提供了ORM的对象关系工具，其中包括JDO、Hibernate和iBatis SQL Map。所有这些都遵从Spring的通用事务和DAO异常层次结构。

■ Spring Web模块:

- Web上下文模块建立在应用程序上下文模块之上，为基于Web的应用程序提供了上下文。所以，Spring框架支持与Struts的集成。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。

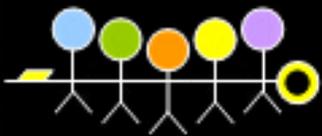
■ Spring MVC框架:

- MVC框架是一个全功能的构建Web应用程序的MVC实现。MVC容纳了大量视图技术，其中包括JSP、Velocity、Tiles、iText和POI。



Spring之依赖反转(IOC)

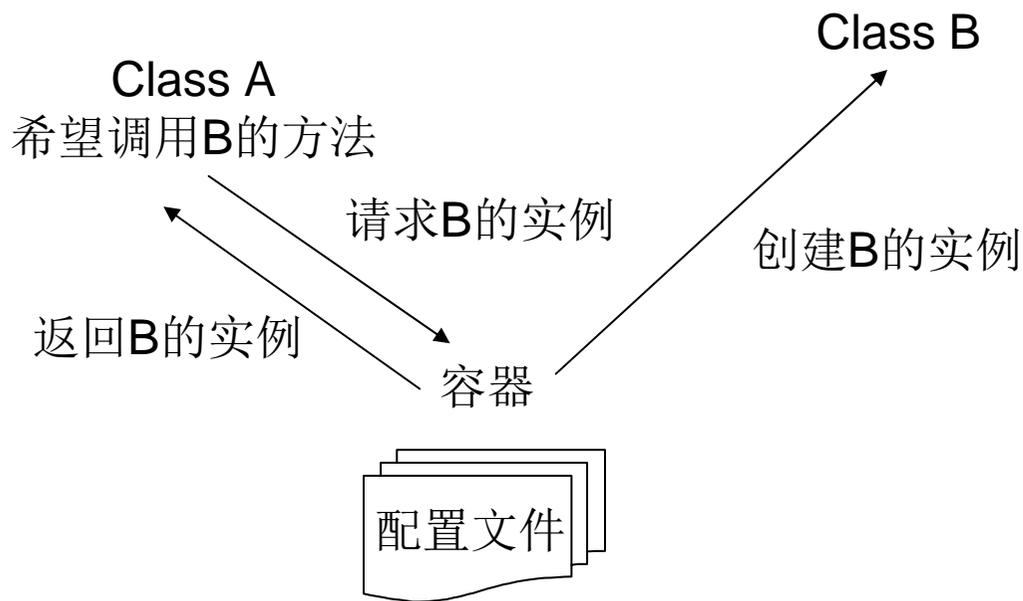
- **IOC的基本思想**：不创建对象，但是描述创建它们的方式。在代码中不直接与对象和服务连接，但在配置文件中描述哪一个组件需要哪一项服务，**Spring**容器负责将这些联系在一起。
- **IOC是工厂模式的升华**，可以把**IOC**看作是一个大工厂，只不过这个大工厂里要生成的对象都是在**XML**文件中给出定义的，然后利用**Java**的“反射”编程，根据**XML**中给出的类名生成相应的对象。
- 从实现来看，**IOC**是把以前在工厂方法里写死的对象生成代码，改变为由**XML**文件来定义，也就是把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性。

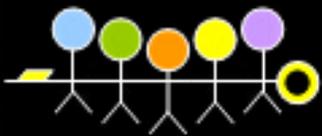


控制反转(Inverse of Control)

■ IOC的基本思想:

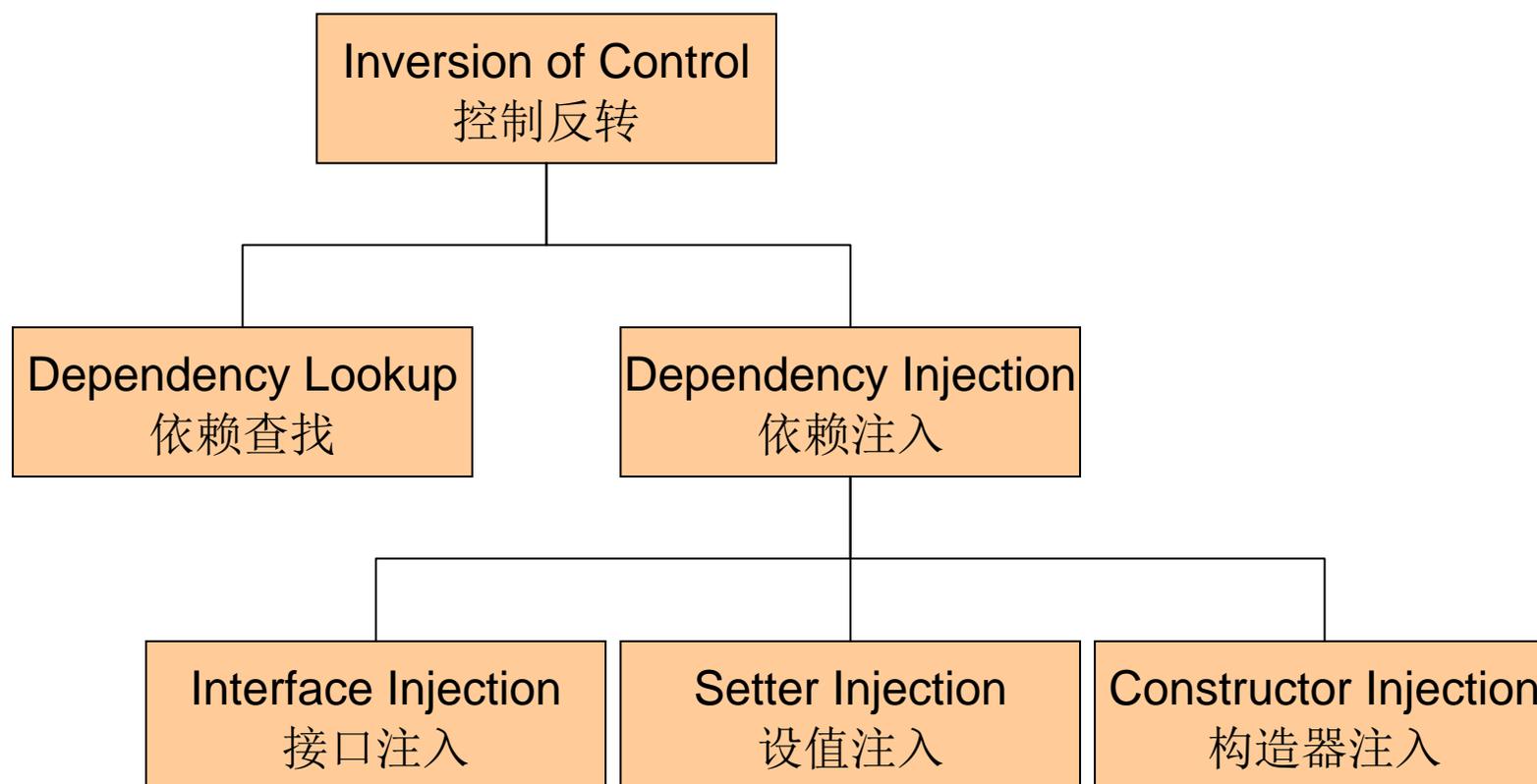
- 由第三方的容器来控制对象之间的依赖关系，而非传统实现中由代码直接操控。
- 控制权由代码中转到了外部容器，带来的好处就是降低了对对象之间的依赖程度，提高灵活性和可维护性。





控制反转(Inverse of Control)

- IoC的实现机制:





IoC实现之依赖查找(Dependency Lookup)

- **依赖查找(Dependency Lookup):** 容器中的受控对象通过容器所提供回调接口和上下文环境(通常为**一组API**)来查找自己所依赖的资源和合作对象。

```
public class MyBusniessObject{
    private DataSource ds;
    private MyCollaborator myCollaborator;

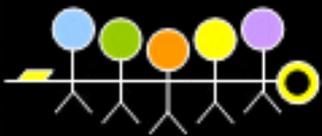
    public MyBusnissObject(){
        Context ctx = null;
        try{
            ctx = new InitialContext();
            ds = (DataSource) ctx.lookup("java:comp/env/dataSourceName");
            myCollaborator = (MyCollaborator) ctx.lookup("java:comp/env/myCollaboratorName");
        }...
    }
}
```

- **依赖查找的主要问题:** 代码必须依赖于容器环境, 所以它不能在容器之外运行。



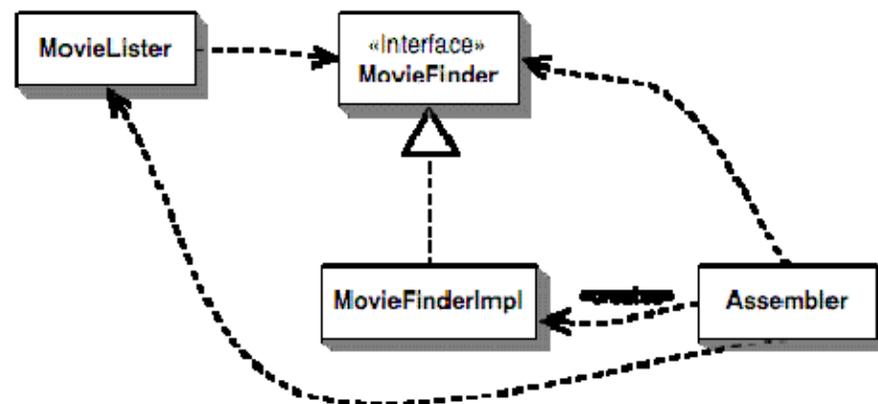
IoC实现之依赖注入(Dependency Injection)

- 依赖注入(Dependency Injection): 受控对象只需要暴露setter()方法或者带参数的构造子(creator)或者接口(interface), 容器全权负责依赖查询, 它会把符合依赖关系的对象传递给需要的对象。
- 与依赖查找方式相比, 依赖注入主要优势为:
 - “查找资源”的逻辑从应用代码中抽取出来, 交给IoC容器负责, 与应用代码完全无关;
 - 不依赖于容器的API, 可以很容易的在任何容器以外使用应用对象;
 - 不需要特殊的接口, 绝大多数对象可以做到完全不必依赖容器。
- <http://martinfowler.com/articles/injection.html>



IoC实现之依赖注入(Dependency Injection)

- **DI的本质特征：**引入第三者(IoC container)，通过事先预定义的配置文件，生成双方实例，并将二者联系起来。



- **DI有三种实现形式：**
 - Type 1: 接口注入(Interface Injection)
 - Type 2: 设值注入(Setter Injection)
 - Type 3: 构造器注入(Constructor Injection)



Spring之面向方面的编程(AOP)

- 面向方面的编程(AOP): 允许程序员对横切关注点(即大量应用程序所通用的横向功能, 例如日志和事务管理)进行模块化。
- AOP的核心构造是“方面”(aspect), 它将那些影响多个类的行为封装到可重用的模块中。
- 例如: 在传统编程模式中, 要将日志记录语句放在所有Java类中才能实现日志功能。
- 在AOP方式中, 可以将日志服务模块化, 并以声明的方式将它们应用到需要日志的组件上。优势就是Java类不需要知道日志服务的存在, 也不需要考虑相关的代码。
- 所以, 用Spring AOP编写的应用程序代码是松散耦合的。



Hibernate: Object-Relation Mapping in J2EE

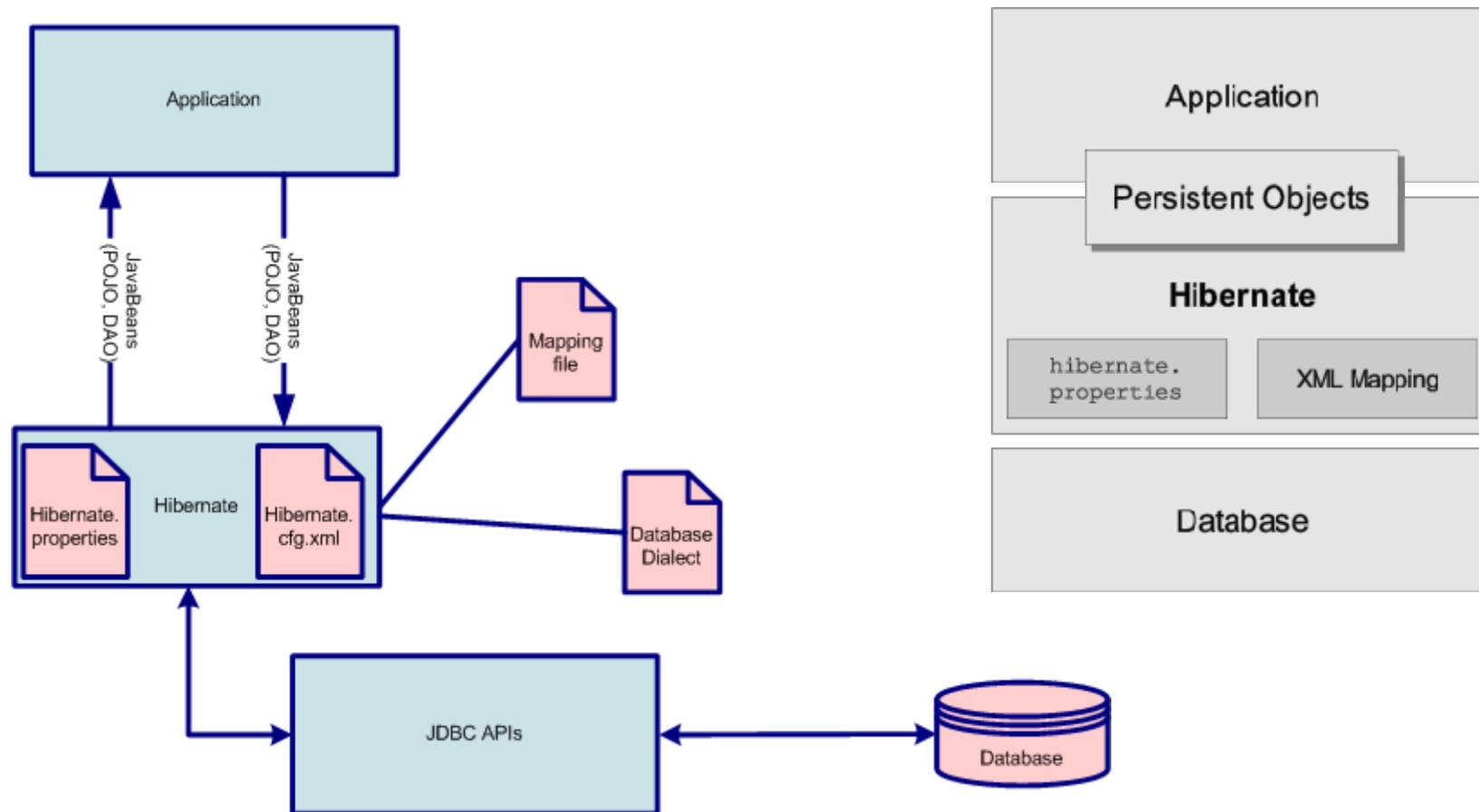


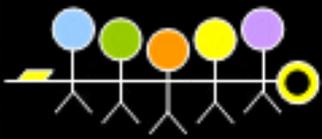
- **Hibernate**是一个开放源代码的对象关系映射框架，它对JDBC进行了非常轻量级的对象封装，使得Java程序员可以随心所欲的使用对象编程思维来操纵数据库。
- 开发J2EE程序的时候，无需考虑后台的关系数据存储，**Hibernate**通过配置完成Java class与JDBC之间的连接。
- <http://www.hibernate.org>



Hibernate: Object-Relation Mapping in J2EE

- **Hibernate**通过基于XML的配置文件来定义映射关系：
 - JAVA类中的属性 \leftrightarrow 关系数据表中的列





Hibernate: Object-Relation Mapping in J2EE

```
public class Message {  
    private Long id;  
    private String text;  
    private Message nextMessage;  
    ...  
    getID();  
    setID();  
    getText();  
    setText();  
    ...  
}
```

关系数据表: Message

消息号(id)	消息内容(text)
1	"Hello World"
2	"This is a test"

向关系表中插入新数据的代码:

```
Session session = getSessionFactory().openSession();  
Transaction tx = session.beginTransaction();  
Message message = new Message("Hello World");  
session.save(message);  
tx.commit();  
session.close();
```

对应的嵌入式SQL语句:

```
insert into Message (id, text) values (1, 'Hello World')
```



Hibernate: Object-Relation Mapping in J2EE

关系数据表: Message

消息号(id)	消息内容(text)
1	"Hello World"
2	"This is a test"

从关系表中查询数据的代码:

```
Session newSession = sessionFactory.openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages = newSession.find("from Message as m order by m.text asc");
System.out.println( messages.size() + " message(s) found:" );
for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
    Message message = (Message) iter.next();
    System.out.println( message.getText() );
}
newTransaction.commit();
newSession.close();
```

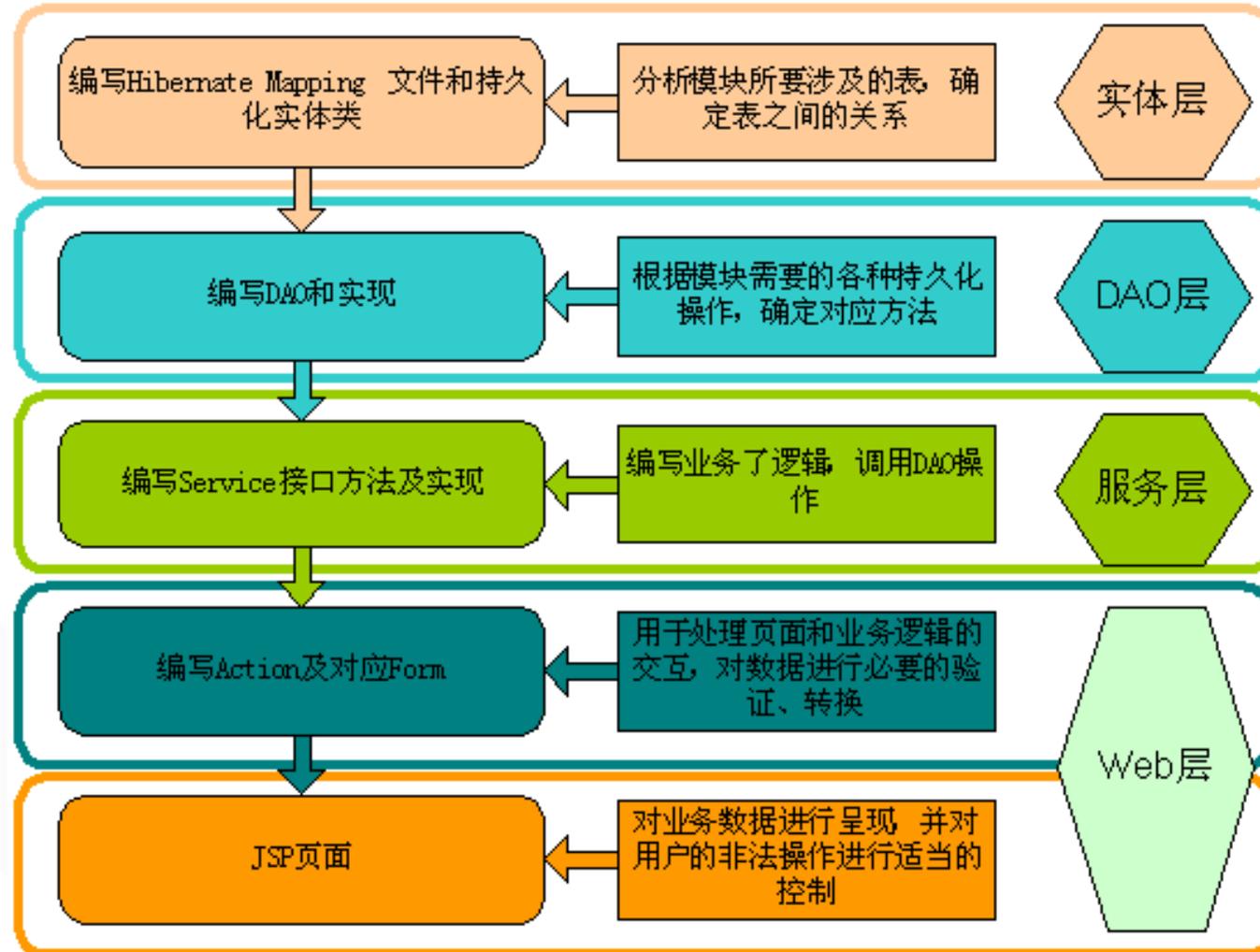
Hibernate Query Language (HQL)

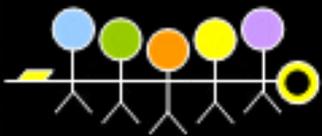
对应的嵌入式SQL语句:

```
select m.id, m.text from Message m order by m.text asc
```



归纳：基于SSH的软件分层架构





3.2 网站架构的演变

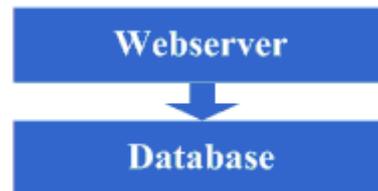
有关该案例的详细内容，请阅读

<http://www.blogjava.net/BlueDavy/archive/2008/09/03/226749.html>



第一步：物理分离web server和数据库

- 二者若驻留在同一台server上，系统的压力越来越高，响应速度越来越慢，数据库和应用互相影响。
- 将应用和数据库从物理上分离，变成了两台机器：

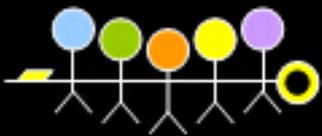




第二步：增加页面缓存

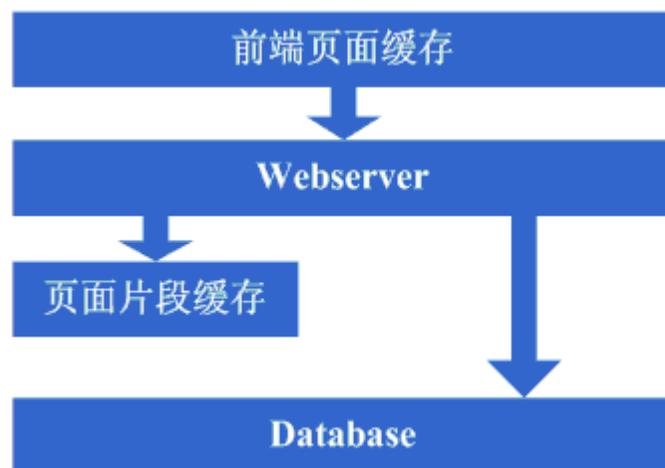
- 随着访问数量越来越多，响应速度又开始变慢。
- 原因：访问数据库的操作太多，导致数据连接竞争激烈；
- 但数据库连接又不能开太多，否则数据库机器压力会很高。
- 考虑采用缓存机制来减少数据库连接资源的竞争和对数据库读的压力，将系统中相对静态的页面进行缓存；
- 这样程序上可以不做修改，就能够很好的减少对Web Server的压力以及减少数据库连接资源的竞争。
- 前端页面缓存技术：例如squid。

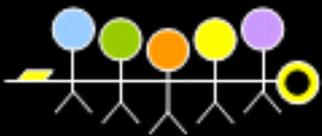




第三步：增加页面片段缓存

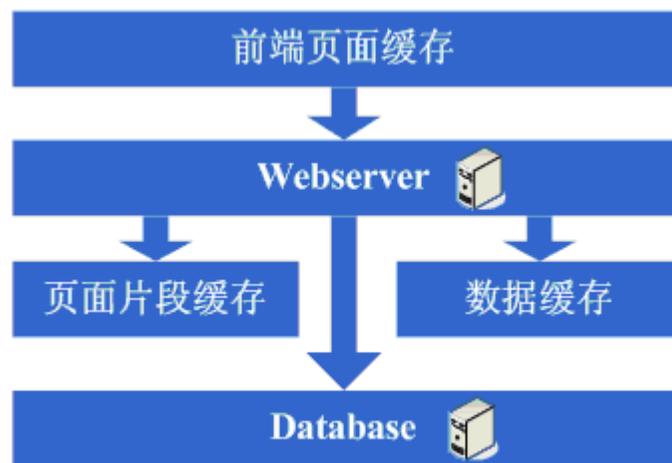
- 随着访问量的增加，系统又开始变慢。
- 除了对纯静态页面进行前端缓存，能否让现动态页面里相对静态的部分也缓存起来？
- 考虑采用页面片段缓存策略，对动态页面中相对静态的片段部分进行缓存。





第四步：数据缓存

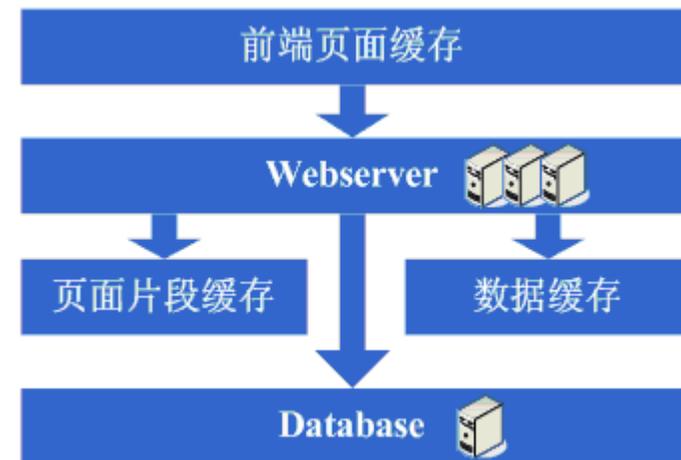
- 随着访问量的增加，系统又变慢。
- 原因：系统中存在一些重复获取数据信息的地方。
- 考虑将这些数据信息也缓存到本地内存。

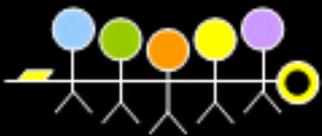




第五步：增加Web Server

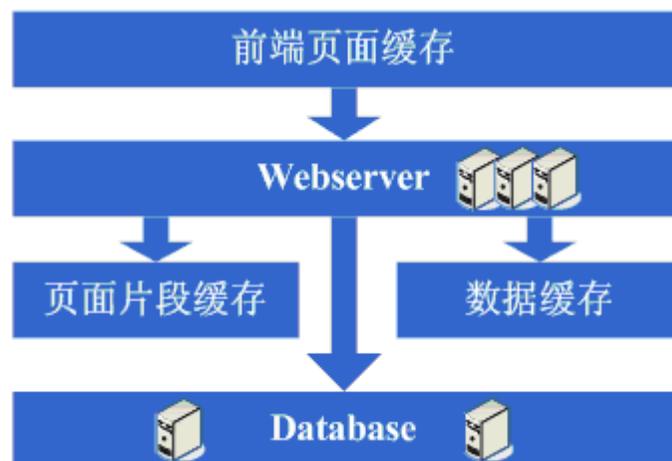
- 随着系统访问量的再度增加，web server的压力在高峰期会上升到很高。
- 考虑增加一台web server，也可避免单台web server宕机后系统的可靠性/可用性NFR。
- 典型问题：
 - 如何让访问分配到这两台机器上：负载均衡；
 - 保持状态信息的同步(例如用户session等)：写入数据库、写入存储、cookie或同步session信息等机制；
 - 保持数据缓存信息的同步(例如之前缓存的用户数据等)：缓存同步或分布式缓存；
 - 如何让上传文件这些类似的功能继续正常：共享文件系统或存储等。

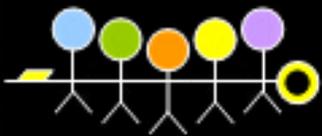




第六步：分库

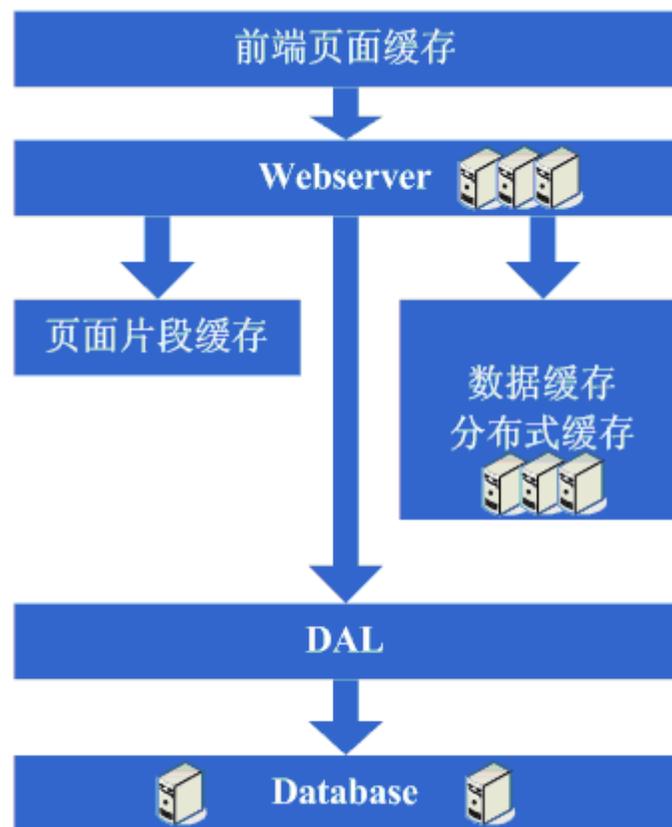
- 系统又开始变慢了...
- 原因：数据库写入、更新的这些操作的部分数据库连接的资源竞争非常激烈，导致了系统变慢。
- 此时可选的方案有：数据库集群、分库策略

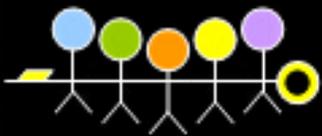




第七步：分表、DAL和分布式缓存

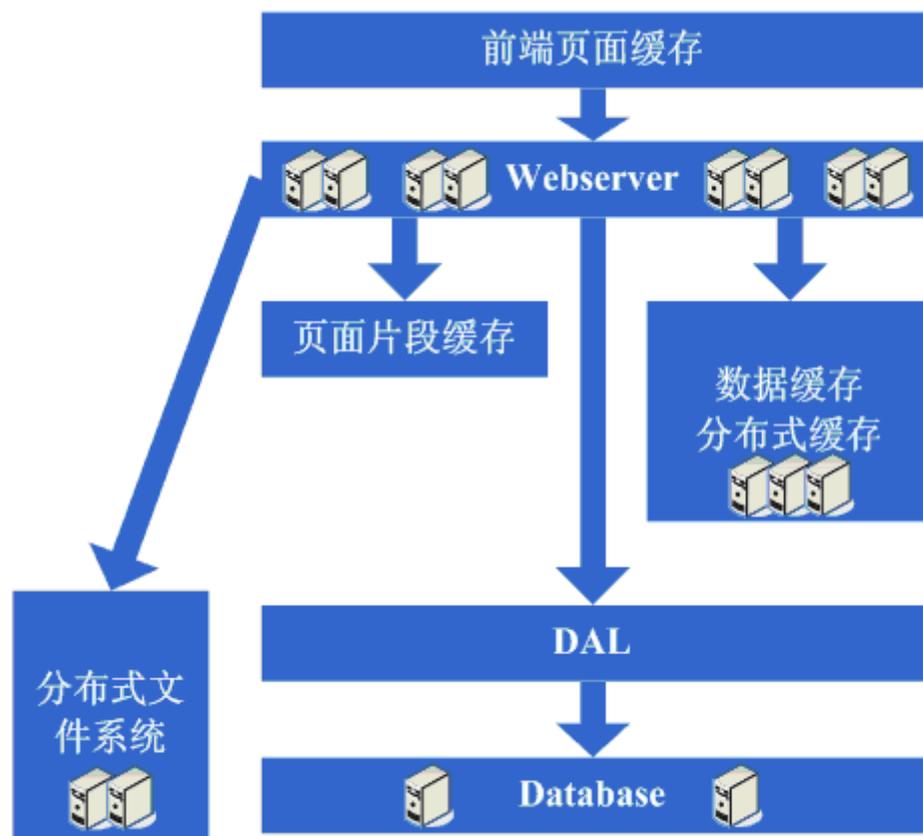
- 随着系统的不断运行，数据量大幅度增长。
- 可按分库的思想开始做分表的工作：分布式数据库、数据分片、分布式缓存。
- 动态hash算法、一致性hash算法。

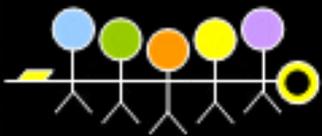




第八步：增加更多的web server

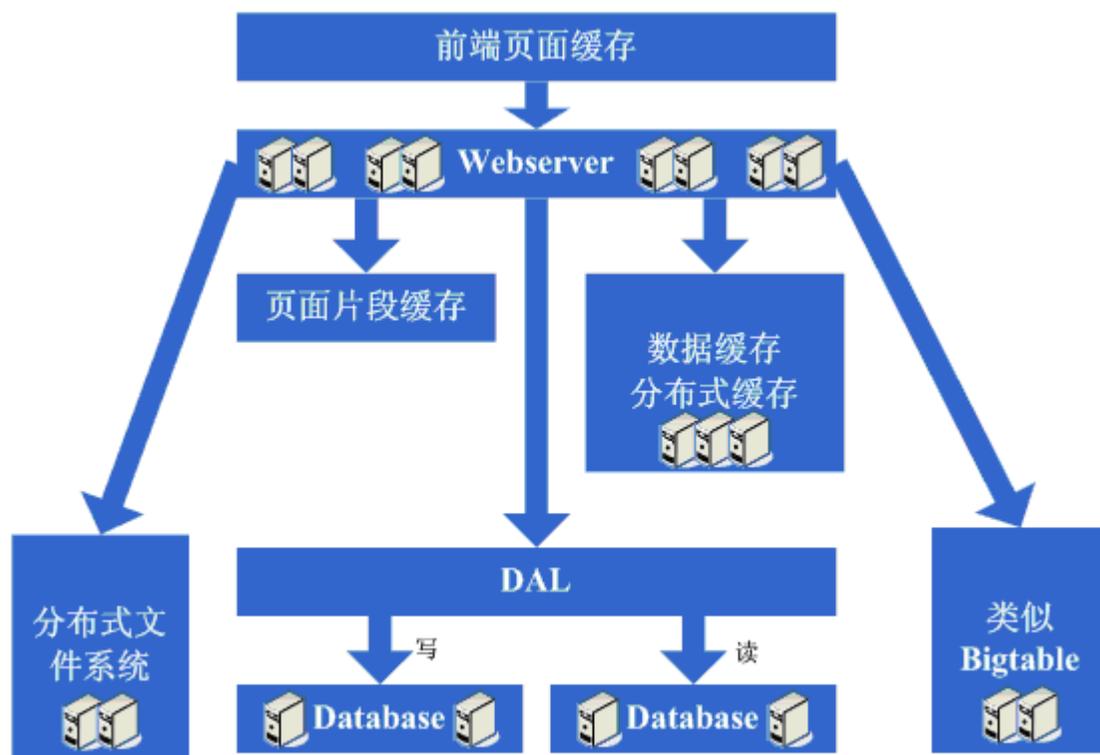
- 系统的访问又开始变慢
- 原因：**web server**阻塞了很多请求，请求数太高导致需要排队等待，响应速度变慢；
- 硬件负载均衡、分布式文件系统
→ **high-Scalability**

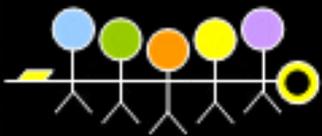




第九步：数据读写分离和廉价存储方案

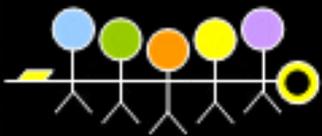
- 分析数据库的压力状况，会发现数据库的读写比很高；
- 通常会想到数据读写分离的方案，同时采用一些更为廉价的存储方案，例如BigTable。





第十步：大型分布式应用时代和廉价服务器群





3.3 大型多人在线游戏(MMO)

有关该案例的详细内容，请阅读《Beautiful Architecture》
第4章(pp. 45-52)



大型多人在线游戏(MMO)

- **MMO = Massively Multiplayer Online Games**
 - 几乎所有玩家在所有时候都在与服务器、与其他玩家交互，并行非常高；
- **设计MMO系统架构的首要需求：必须具备伸缩性，以满足大量的玩家，且玩家的性能体验(流畅性)不能降低。**
- **关于MMO的几个特征：**
 - 针对伸缩性而设计的架构，通常使用多台server构成集群，通过并发完成大量请求；当玩家数量增加时，通过扩展集群的规模来保证性能；
 - 超胖的客户端+相对很瘦的服务器：与传统web应用相反(瘦客户端+胖服务器)；
 - 客户端任务只访问服务器上的少量状态数据，但其中50%被改写：与传统web应用相反(90%的数据访问是只读的，只改写少量数据)；
 - 当客户端性能和吞吐量之间产生矛盾时，应尽可能减少延迟，哪怕以牺牲吞吐量为代价。



对MMO架构设计的基本认识(1)

- **基本认识1：**所有针对伸缩性而设计的架构，通常需要包含多台机器；
 - 从很小的系统开始，随着用户数的增长而增加处理能力；
 - 采用多核CPU，各核通过并发来实现系统总体性能的增长；
 - MMO的总体架构应采用分布式和并发的系统架构。
- **对MMO程序员来说，开发分布式并发程序过于复杂了**
 - 程序员把系统视为一台单机，运行着一个线程；
 - 对应用程序隐藏分布式和并发，而是由基础设施来考虑；
- **基本认识2：**MMO的编程模型应是事件驱动的
 - 服务器端部署事件监听器，监听客户端生成的事件；
 - 如果检测到事件，服务器生成一项任务并执行；
 - 服务器也可周期性的生成任务，不受客户端玩家的控制；



对MMO架构设计的基本认识(2)

- **基本认识3: MMO的架构与传统企业架构的区别**
 - 传统的企业应用的架构: 瘦客户端+胖服务器;
 - MMO的架构: 超胖的客户端+超瘦的服务器

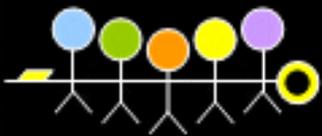
 - MMO对客户端计算机CPU速度、内存和图形性能的要求极高, 专门处理图形密集的、高度交互的任务; 只要有可能, 数据就存储在客户端;
 - 服务器端则尽可能的保持简洁, 仅保存共享的世界真实状态, 在不同玩家之间保持一致, 避免作弊。

- **基本认识4: 数据访问模式的不同**
 - 传统企业应用中: 90%的数据访问是只读的, 只改写少量数据;
 - MMO中: 大多数客户端任务只访问服务器上的少量状态数据, 但其中50%被改写。

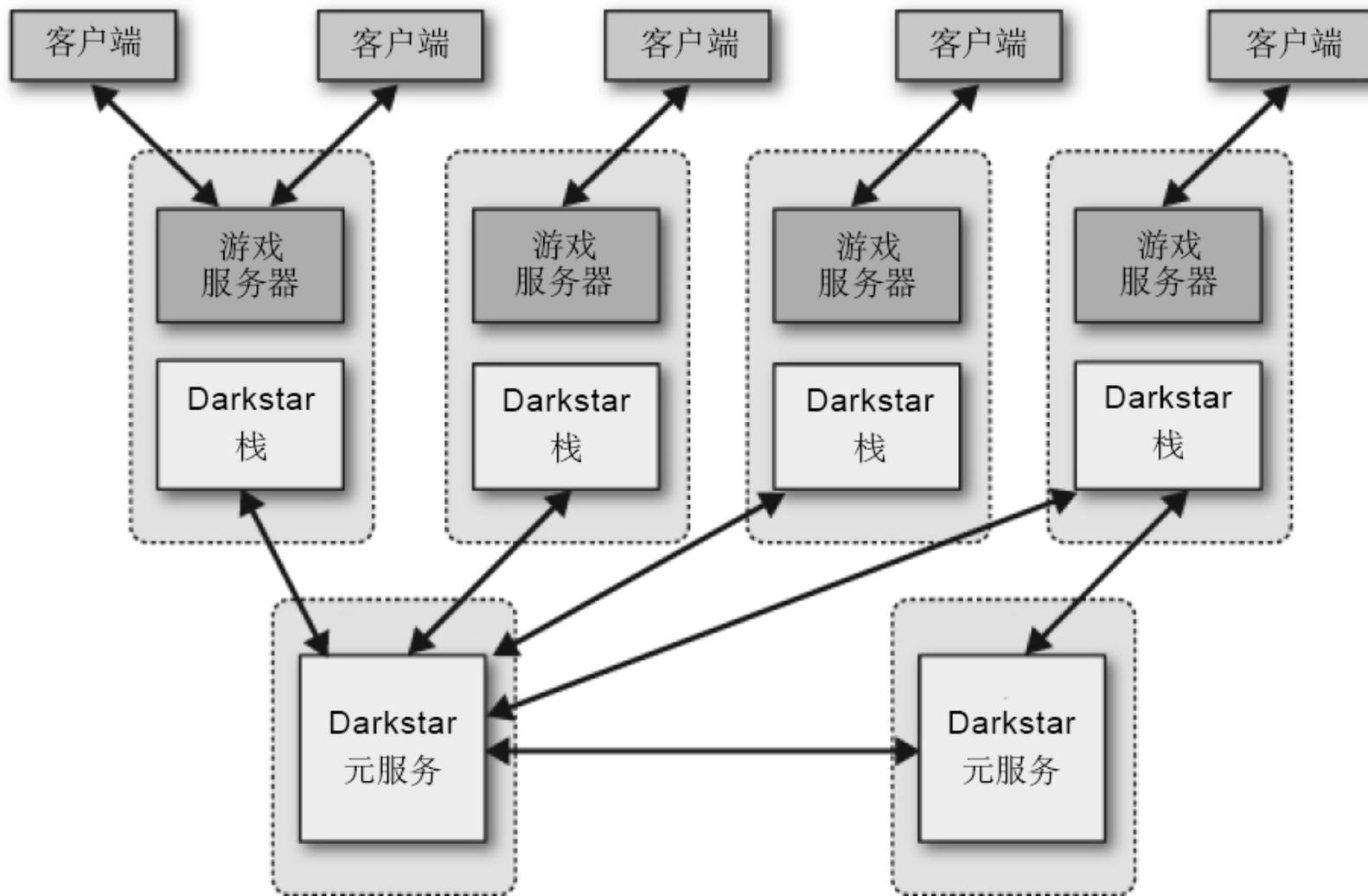


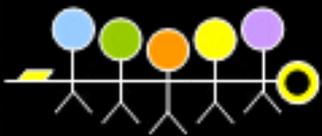
对MMO架构设计的基本认识(3)

- **基本认识5：延迟和吞吐量**
 - MMO中应尽可能减少延迟，哪怕以牺牲吞吐量为代价；
- **降低延迟、提高伸缩性的设计方案：**
 - 将游戏划分成不同的区域，将不同的区域分配给不同的服务器；
 - 对某些热门的拥塞区域，采用sharding的策略：为该区域复制多个副本，每个副本运行在自己的服务器上，独立于其他的副本。
 - — 提高了吞吐率，降低了延迟，但处于不同shards上的玩家之间无法交互。
 - — 而且，如何划分区域是直接写在了游戏源代码里的。
- **设计目标：MMO应做到随时伸缩，但游戏逻辑不能受到伸缩的影响。**



Darkstar项目的高层架构设计





Darkstar项目的高层架构设计

- 游戏服务器不需要知道其他服务器的存在；
- 不同服务器之间的协作由基础设施来完成；
- 客户端通过直接通道或“发布-订阅”的机制与服务器端通讯；
- 各栈由一组元服务来协调，对程序员不可见，支持各个副本之间的协作：
 - 如果某个副本失效，就进行失效恢复；
 - 监控各副本负载，必要时重新分配负载，或随时添加新的服务器。





Darkstar的基础服务

- **数据服务(data service):**
 - 保存、读取和操作所有的持久数据;
 - 将多个副本的数据联系在一起, 保持同步;
 - 使用传统的数据库是否恰当?
- **任务服务(task service):**
 - 调度或执行任务: 响应客户端的事件, 或是服务器内部发起的逻辑;
 - 任务从“数据服务”中读取数据, 操作, 通信, 结束;
 - 因为任务多, 并发性高, 底层的“数据服务”要检查任务之间对数据访问的冲突, 确保事务性和一致性;



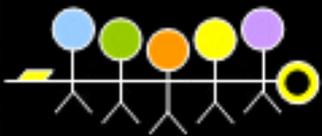
Darkstar的基础服务

- **客户端会话服务(session service):**
 - 客户端和服务端之间通信的中介;
 - 确保维持消息的顺序, 从而使得任务服务可以并发的处理收到的任务;
 - 隐藏了客户端和服务端的真实地址;
- **通道服务(channel service):**
 - 一对多的通信机制, 允许多个客户端之间直接通信, 避免了服务端端的负载。
 - 但是, 为了保证安全性, 所有通道消息需要经过服务器的检查和确认;
- **对伸缩性的支持:** 将服务器通信的端点从一台机器移动到另一台机器, 同时不会改变客户端对这次通信的感觉, 从而做到负载均衡和动态伸缩。



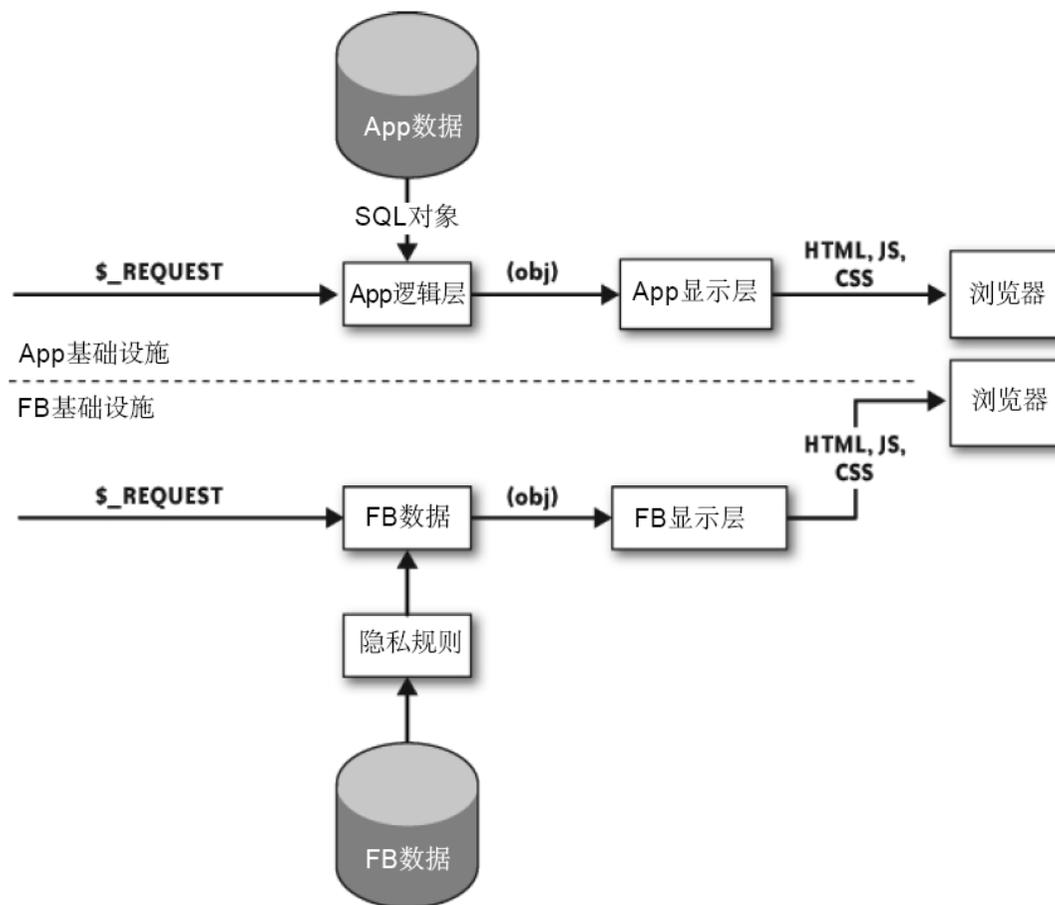
3.4 Facebook的开放API架构

有关该案例的详细内容，请阅读《Beautiful Architecture》
第6章(pp. 111-152)



Facebook数据的对外开放与集成

- Facebook是一个著名的social networking网站；
- 用户可直接登录facebook访问自己的相关数据；



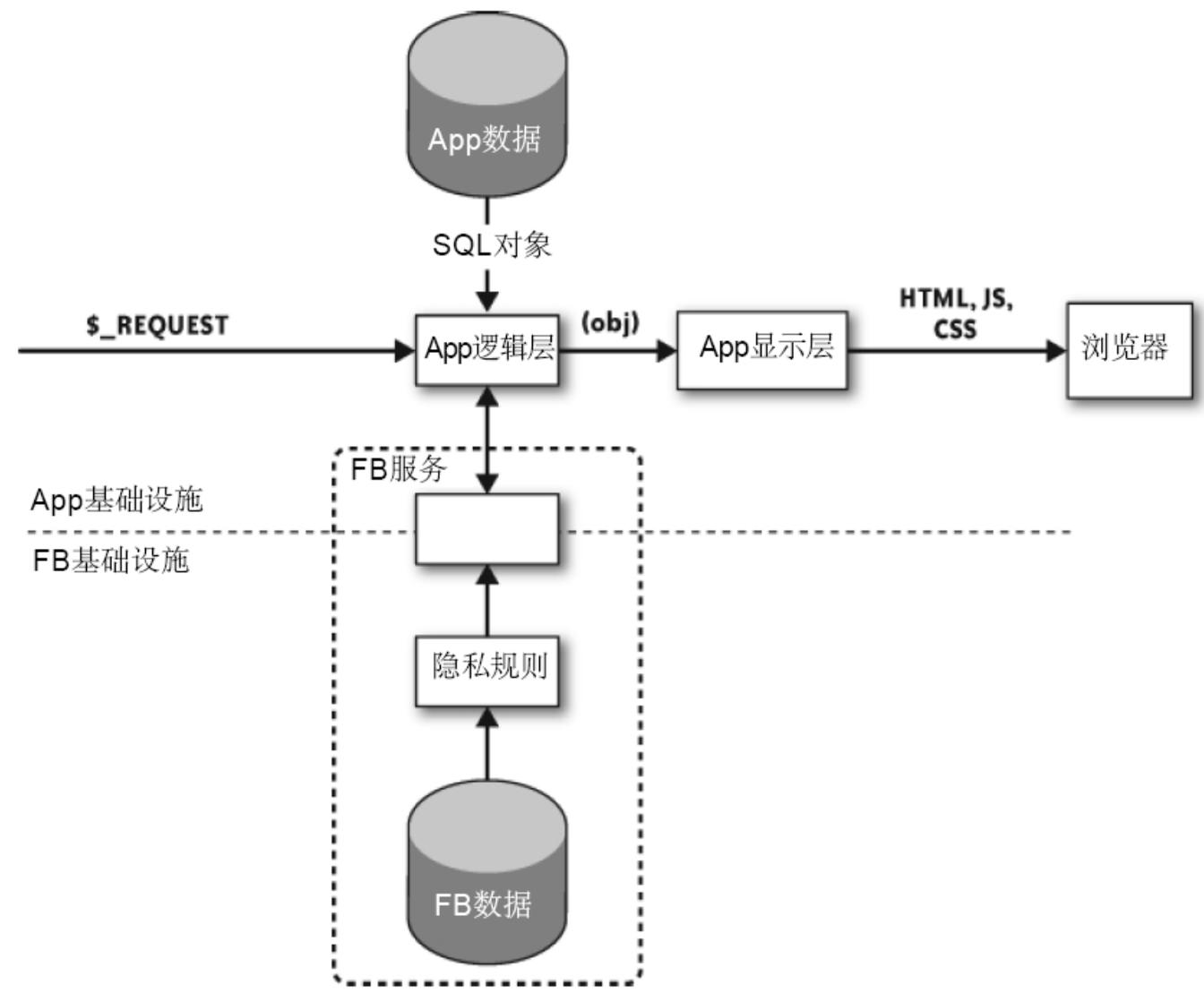


Facebook数据的对外开放与集成

- 解决方案：
 - 将数据功能从facebook内部移到对外开放的web service上(Facebook API);
 - 授权访问这个web service, 保持数据的安全性与隐私性;
 - 创建一种新的数据查询语言(Facebook FQL), 减轻因为引入新的web服务客户端对Facebook本身带来的额外负担;
 - 创建一种新的数据驱动的标记语言(Facebook FBML), 将外部应用对数据的变更集成回Facebook。



基于web service的数据开放(Facebook API)

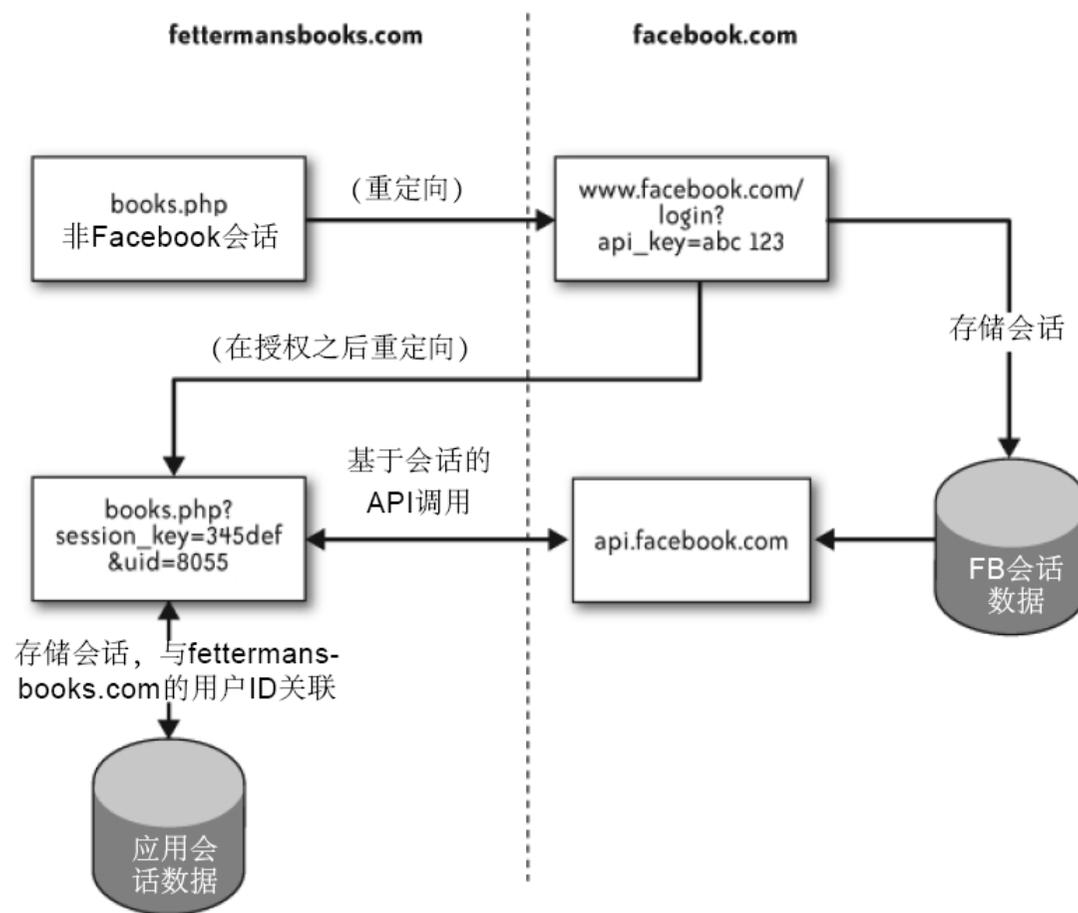


- 标准的Web Service形式API
- 基于SOAP协议的消息传递
- 基于XML Schema的数据表示



Facebook API的认证访问架构

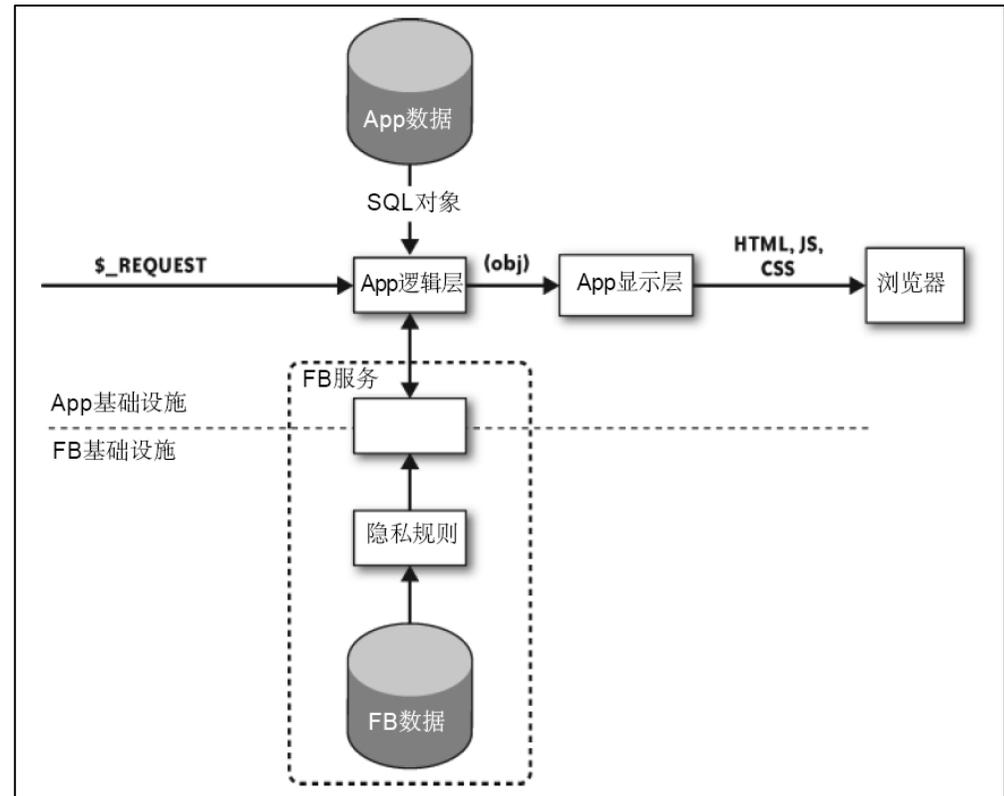
- 若直接访问facebook网站，用户可手工输入ID和password，形成一个session，并通过浏览器cookie保存，在每次请求数据时使用；
- 在Facebook API中，如何实现？
 - 用户通过一个已知的api_key重定向到Facebook登录界面。
 - 用户在Facebook上输入口令，对这个应用授权。
 - 用户带着会话键和用户ID重定向到已知的应用。
 - 应用现在获得了授权，可以代表用户调用API方法。





Facebook的外部数据查询语言(FQL)

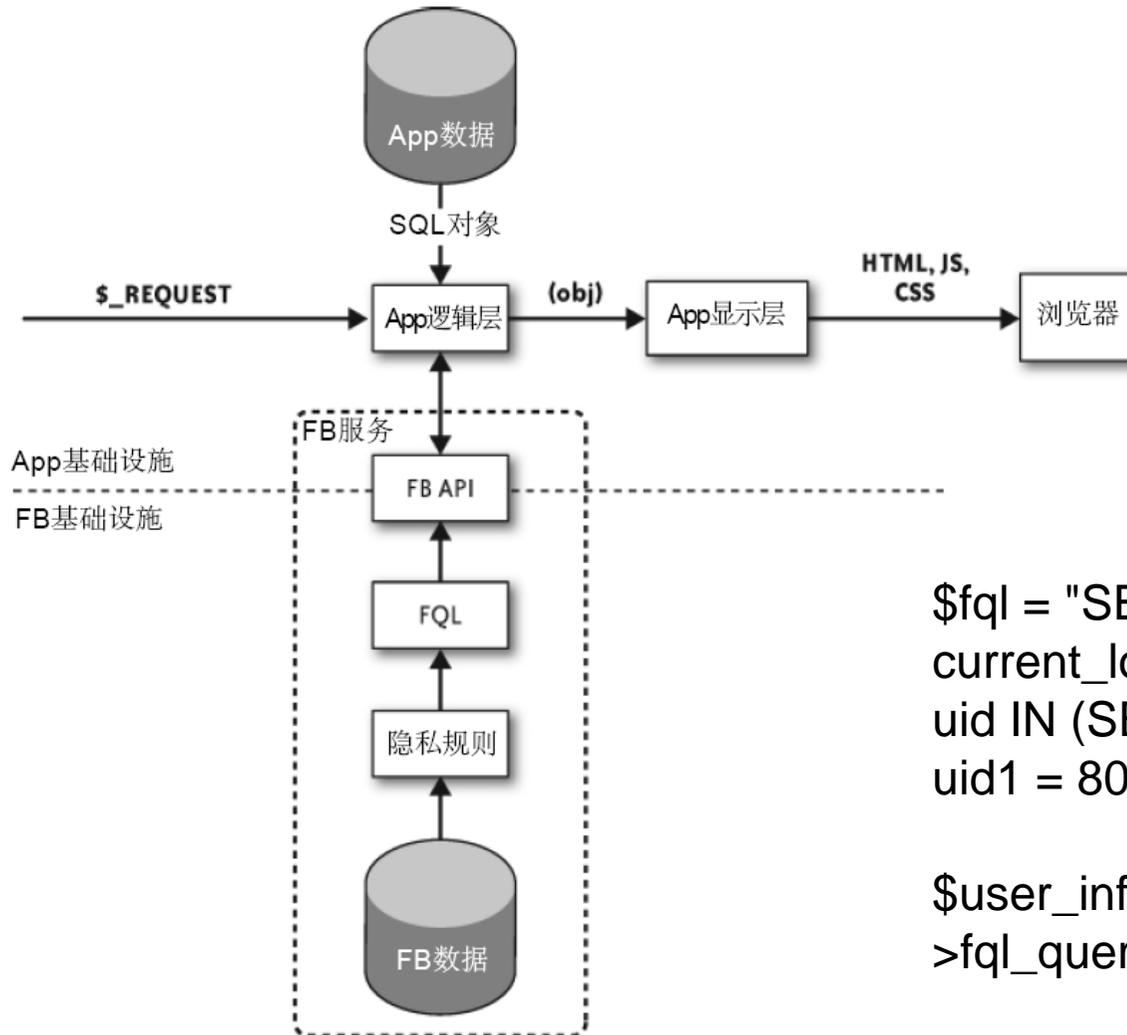
- 存在的问题：外部应用通过WS-based API访问Facebook数据，相比于它访问自己内部的数据，其开销很巨大；
- 有没有办法降低这个开销？——类似内部数据采用的SQL模式，实现外部数据访问模式。
- Facebook的解决方案：FQL
 - 类似于SQL；
 - 将Facebook的内部数据转换为table和fields，而不是传统WS返回的XML schema；
 - 从而，外部应用开发者可像使用SQL一样查询Facebook内部数据。





Facebook的外部数据查询语言(FQL)

- <http://developers.facebook.com/docs/reference/fql/>



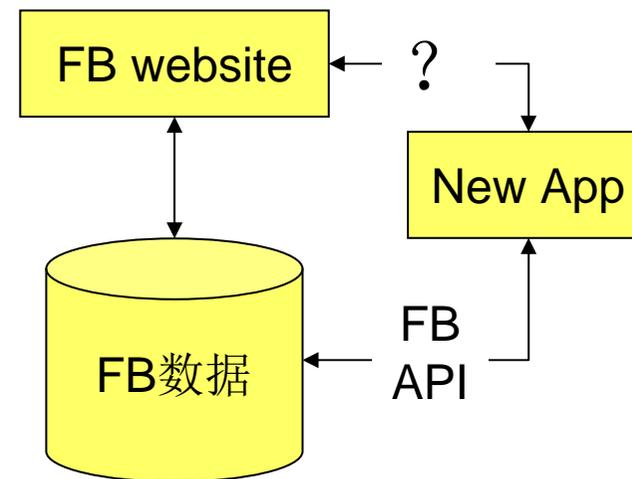
```
$fql = "SELECT uid, name, books, pic,
current_location FROM profile WHERE
uid IN (SELECT uid2 from friends where
uid1 = 8055)";
```

```
$user_infos = $facebook->api_client-
>fql_query($fql);
```



更深层次的集成：FBML

- **新问题1：**不应仅仅是数据层面的交互，是否可以集成UI层面——使facebook用户也可以在facebook网站上使用这些外部应用并进行交互
- **目的：**增加表现力，留住用户；
- **新问题2：**外部应用如何充分利用Facebook API没有暴露出来的哪些核心数据(例如隐私信息)?
- **目的：**更强大的应用能力；





尝试：Facebook直接通过URL连接到外部应用

- 假设新应用的URL是<http://fettermanbooks.com>
- 那么，对http://apps.facebook.com/fettermansbooks/path?query_string=XXX的请求将被facebook转移到http://fettermanbooks.com/query_string=XXX上；
- Facebook只是简单的读取新应用URL的内容并在facebook主页面中展示；
- 优点：简单；
- 缺点：不安全(类似于代码或脚本注入)、未有本质上的集成。



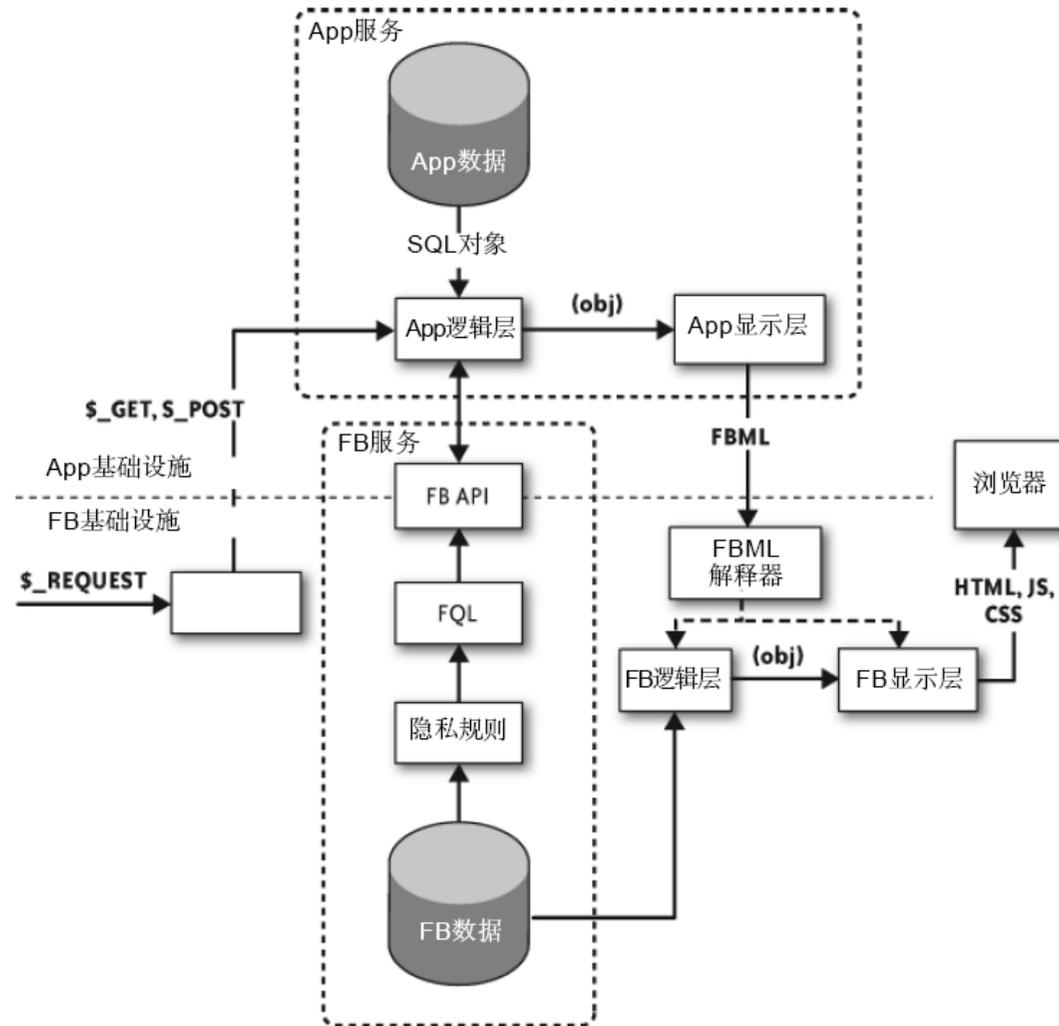
Facebook的解决方案：FBML

- **Facebook的解决方案：**应用开发者通过一种数据驱动的标记语言，在应用中创建应用执行和显示的内容，并与**Facebook**交互。
- **基本思想：**
 - 外部应用发给facebook的不是html代码，而是一种特定的标记语言，其中定义了足够的标记来表现其逻辑和显示，也包含了对facebook受保护数据的请求，完全让facebook在受信任的服务器环境中处理并显示它。
- 这种语言称为**FBML (Facebook Markup Language)**。



Facebook的解决方案：FBML

- <http://developers.facebook.com/docs/reference/fbml/>





Facebook的其他支持功能

- 引入的额外困境1:
 - 在外部应用中，本来由浏览器保存着用户的cookie信息，但现在这种cookie不可用了
 - 因为应用的客户端变成了Facebook平台而非用户的浏览器(注意：是由facebook通过FBML的形式来使用外部应用了)；
- 如何解决？
 - 让Facebook具有浏览器的职责，在Facebook自己的存储库中复制这种cookie功能。
 - 如果应用的FBML服务送回请求头，试图设置浏览器cookie，Facebook就保存这个cookie信息，以(user, application_id)对为主键。
 - Facebook然后“重新创建”这些cookie，就像用户向这个应用栈发出后续请求时浏览器所做的一样。

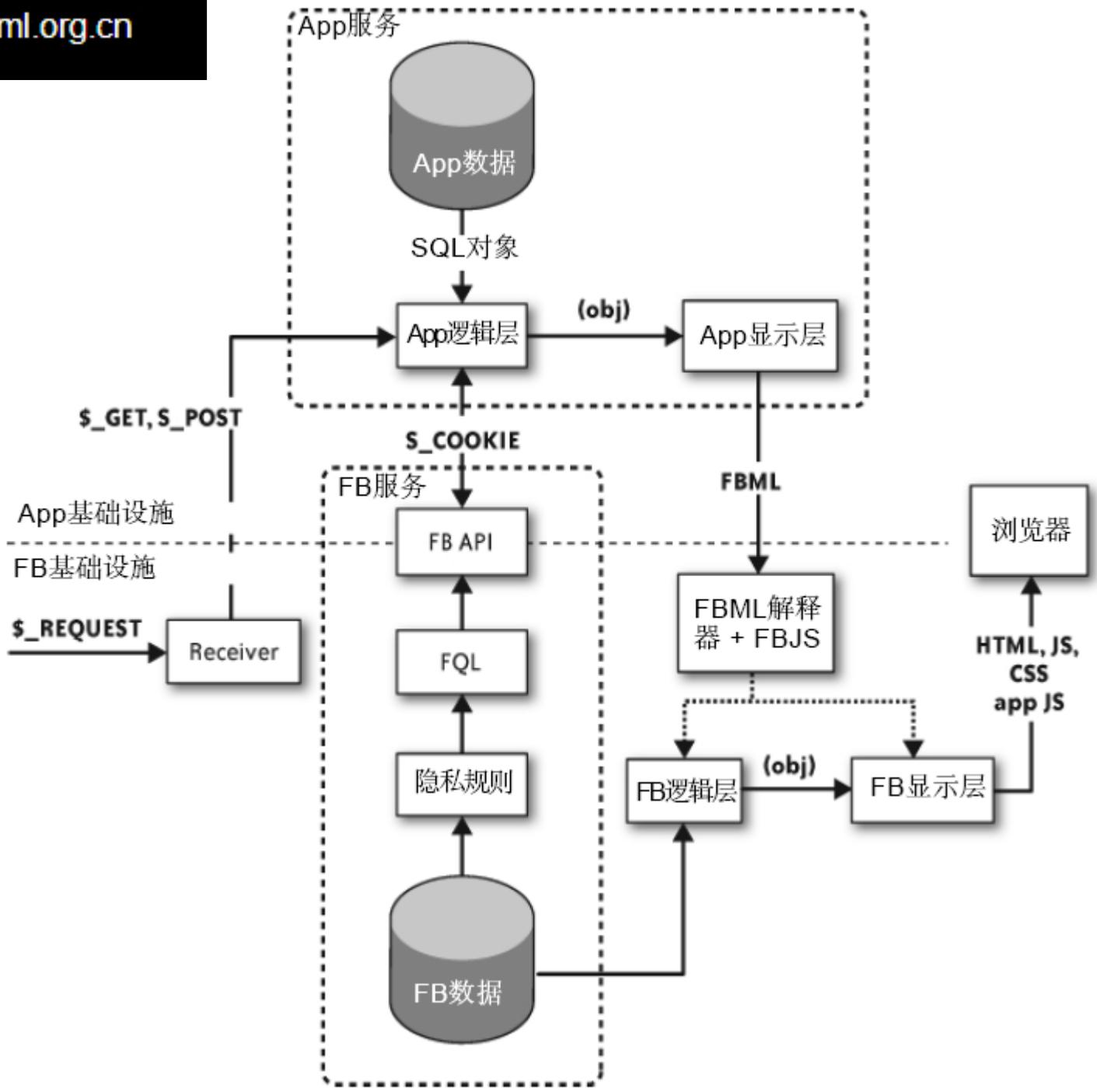


Facebook的其他支持功能

- 引入的额外困境2:
 - 外部应用被facebook所调用，就没有机会执行原本在用户浏览器里执行的JavaScript脚本；——原因：受到脚本或代码注入的威胁，facebook对其禁用了；
 - 这会排除很多有用的功能，例如AJAX或者在不刷新的情况下动态操作页面内容；
- 如何解决？
 - Facebook提供了一些受限的JavaScript库，称为FBJS



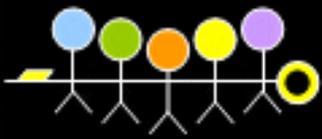
归纳





3.5 面向服务的移动应用架构

H.J. La, S.D. Kim. Balanced MVC Architecture for Developing Service-based Mobile Applications. 2010 IEEE International Conference on E-Business Engineering, Shanghai, China, Nov. 10-12, 2010. 292-299.



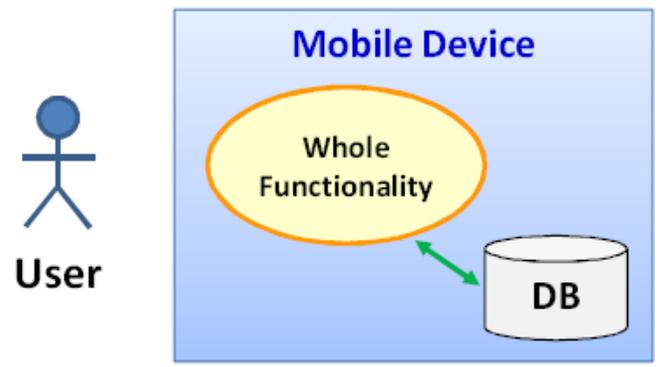
面向服务的移动应用(SMA)的特征

- 随时随地通过Wi-Fi、3G/4G等移动网络访问服务；
 - 感知用户场景(位置、光线强度、周围其他用户等)；
 - 移动网络的不稳定特性使得服务访问不稳定；
 - 客户端的计算和存储能力限制；
 - 客户端的电源限制。
- 那么，SMA的架构应如何设计？

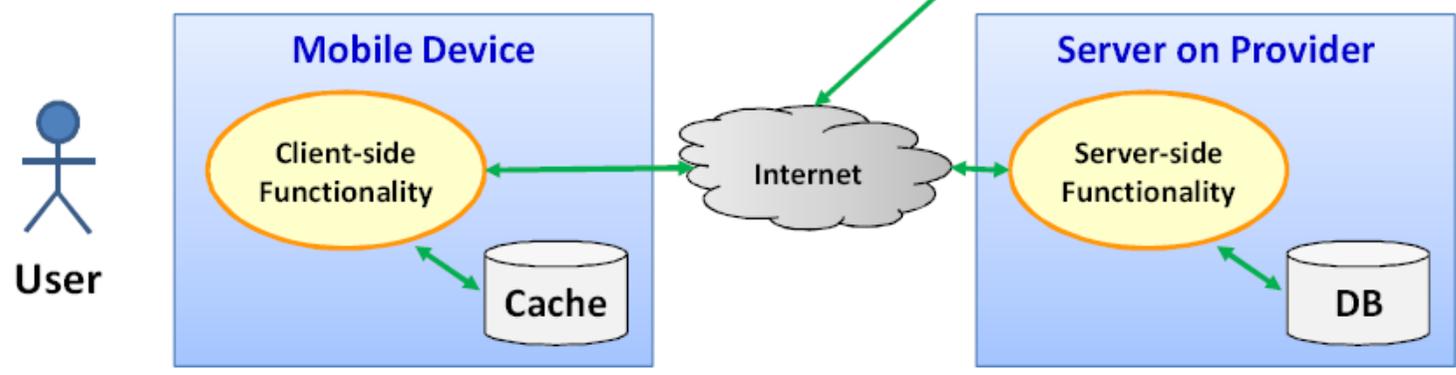


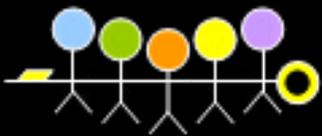
两种类型的SMA架构

Standalone Mobile Application



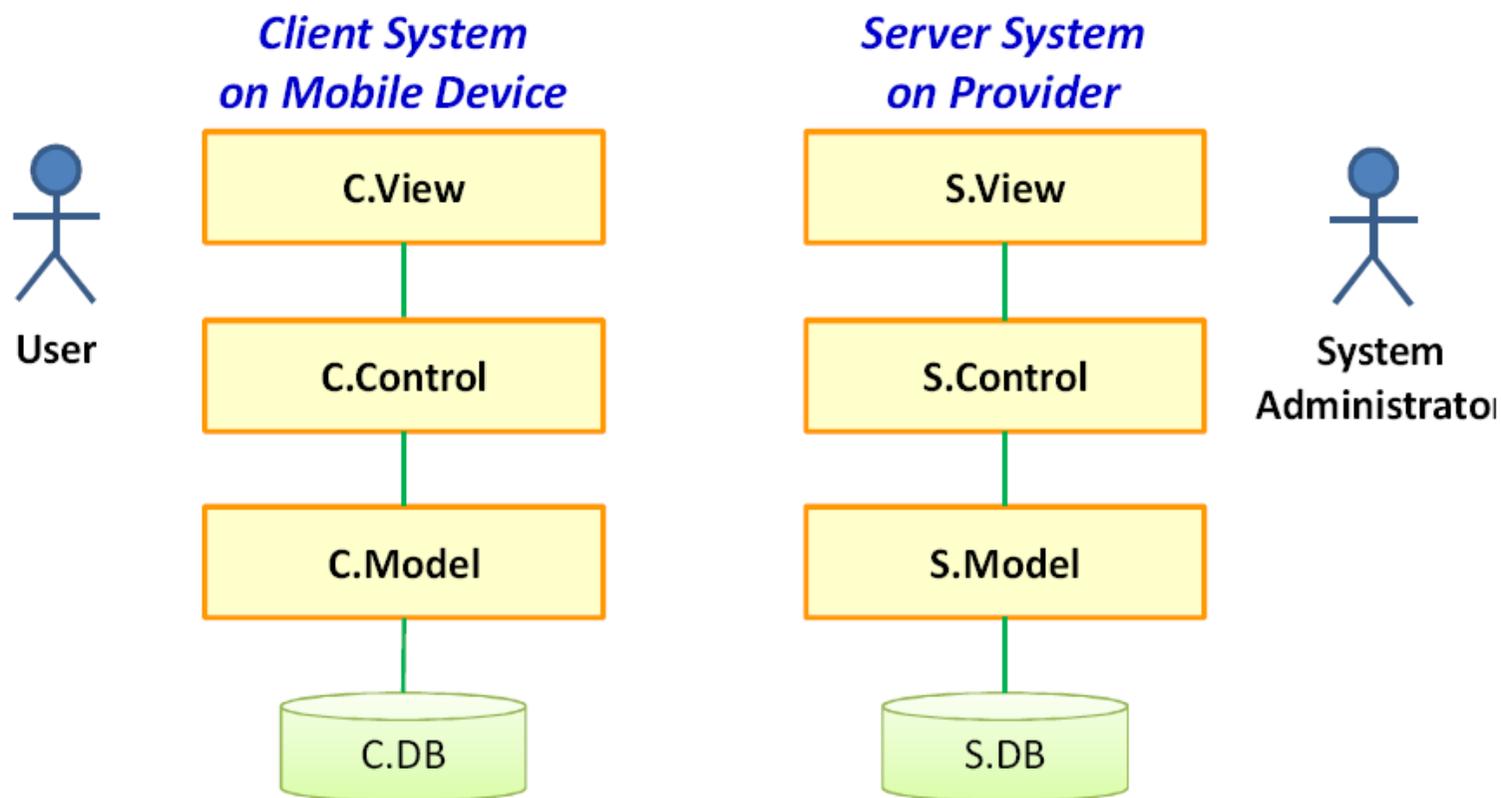
Service-based Mobile Application





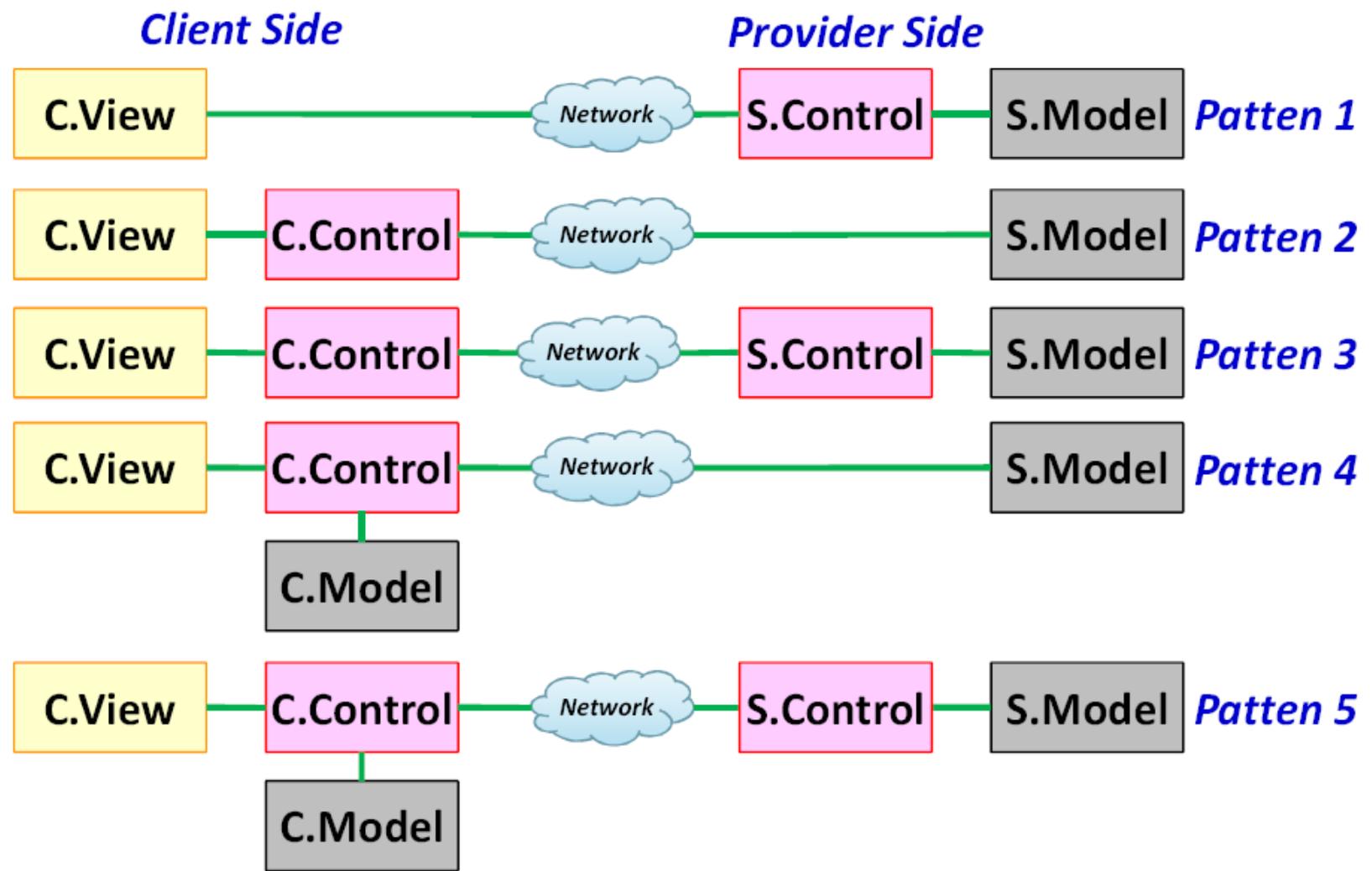
如果采用MVC架构设计SMA

- 核心问题：如何划分在mobile device和server上的功能？





五种可能的划分策略



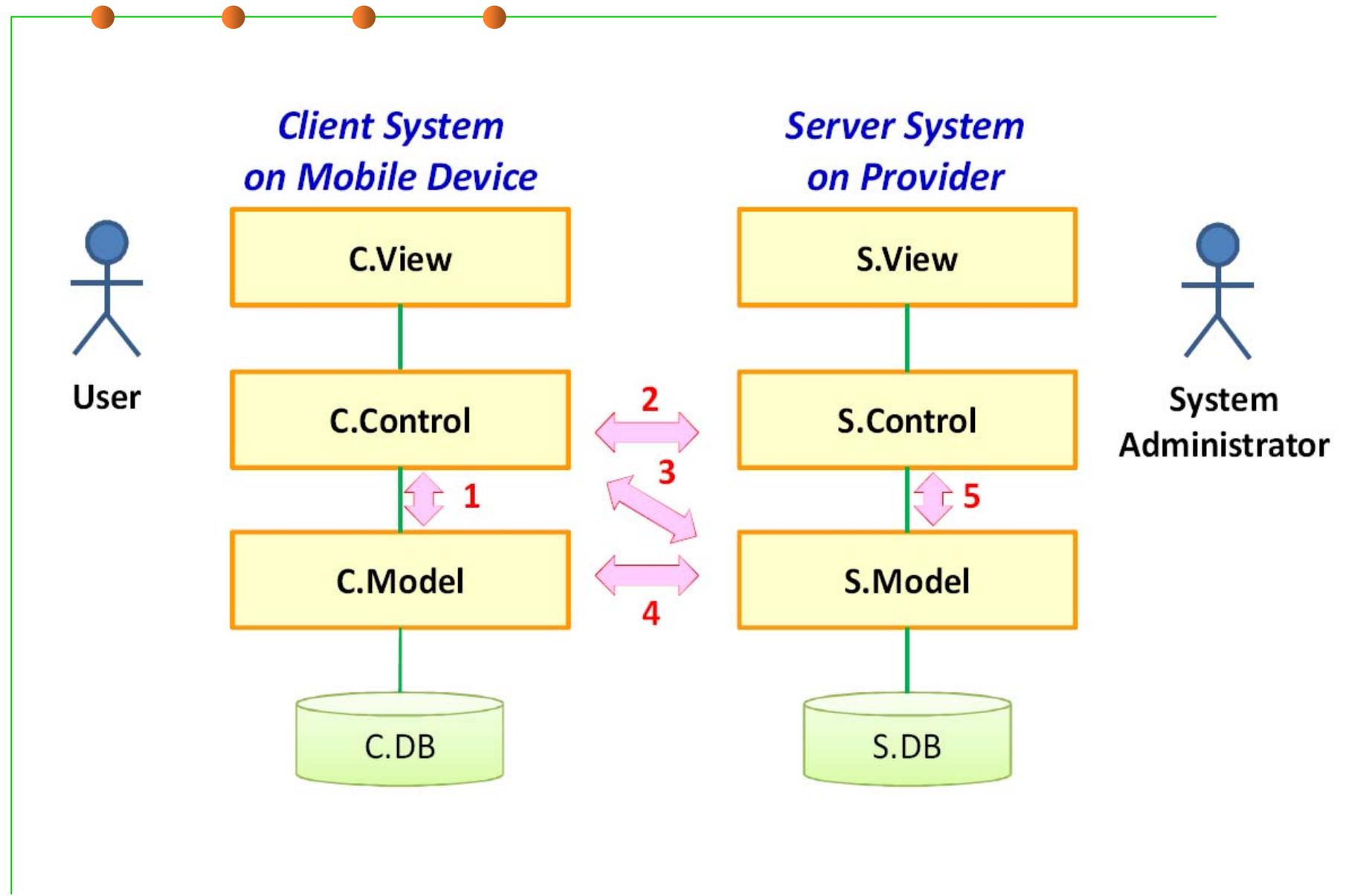


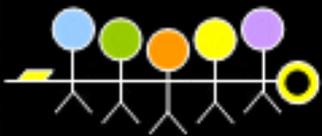
五种策略之间的性能对比

Pattern	Computation on Client Side	Network Overhead	Parallelism
1	Low	Low	Low
2	Middle	High	Low
3	Middle	Low	High
4	High	High	Middle
5	High	Low	High



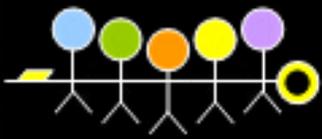
可能的交互途径





3.6 微博的消息架构分析





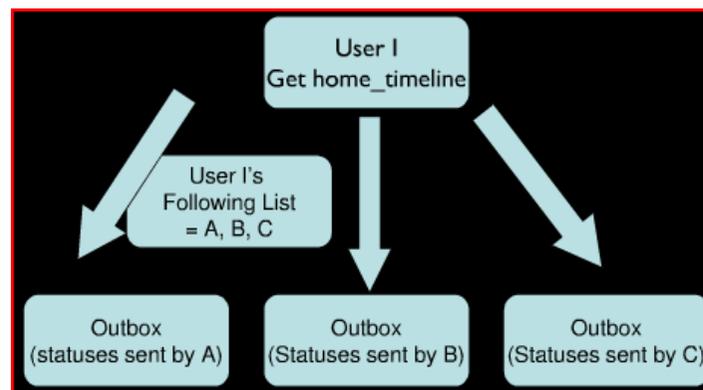
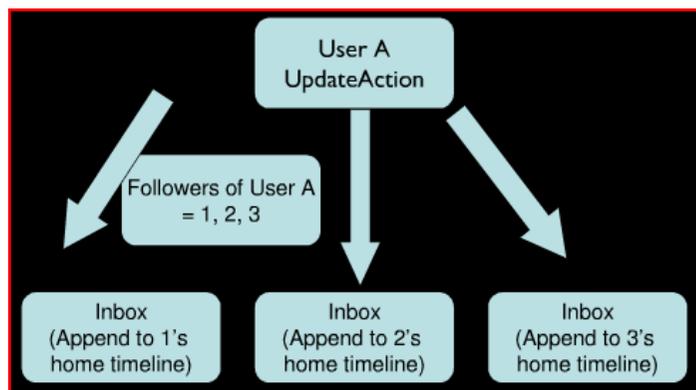
服务架构

- 较之传统的企业应用(ERP等), 微博等SNS新应用对NFR的要求更高了。
- 大规模并发请求、数据流量大(视频、图片、文字、...)
 - ➔对服务器能力(I/O吞吐量)要求高
 - ➔客户端性能(延迟/实时性)要求高
- 需求波动强烈
 - ➔对伸缩性要求高
- 7×24×365
 - ➔可靠性要求高, “永不宕机的服务器”
- - ➔运维成本要低



案例分析：微博信息聚合模式

- 从架构上来分析，微博需要解决的是消息分发问题。
 - 传统的架构模式：事件驱动方式(P2P，观察者模式)
 - 把微博看作邮件：Inbox(收到的微博)、Outbox(已发表微博)
- **PUSH模式**
 - 发表：存到所有粉丝的inbox中(重)
 - 查看：直接访问Inbox(轻)
 - 优点：实现简单；
 - 缺点：分发量大，浪费多。
- **PULL模式**
 - 发表：存到自己的outbox中(轻)
 - 查看：所有关注对象的Inbox(重)
 - 优点：节约存储；
 - 缺点：计算量大。

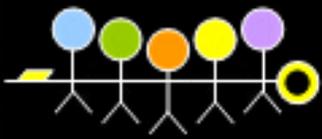




通过异步机制提升性能，牺牲实时性

- 平均500条/秒 → 平均10000条/秒?
 - Latency
 - DB read timeout
 - 前端timeout (503 error)
- 方案：异步机制
 - 不同步等待
 - 将消息存入消息队列(message queue)
 - 轻量级的发表
- MemCacheQ: Simple queue service over MemCache
 - Get/Set
 - 把某些耗时的工作推后，在后台慢慢地去执行，这样就不会让用户等待太久。
- 使用多个MemCacheQ服务器以避免单点故障
 - Get操作：轮询所有服务器
 - Set操作：随机选择一个

“任何地方都要异步，在每个环节都能异步，通过细分打破串行化”



实时性

- MemCacheQ的异步方式降低了等待时间，但牺牲了一部分实时性。
- 解决方法：Cache中心化，“让数据离CPU最近，避免磁盘的IO”
 - 页面缓存：Squid
 - 内存缓存：MemCached
 - 数据库缓存：Database buffer / cache
- Cache的容量问题：过大、过小都不好；
 - 压缩(QuickLZ/LZO)→Get性能下降，Set性能提升，空间降低；
- 分布式Cache：按策略将不同对象分配到不同的memcached实例上，通过Consistent Hash机制减少cache节点变动带来的缓存失效；
- Cache替换策略：Read-Through、Write-Through、Refresh-Ahead、Write-Behind。

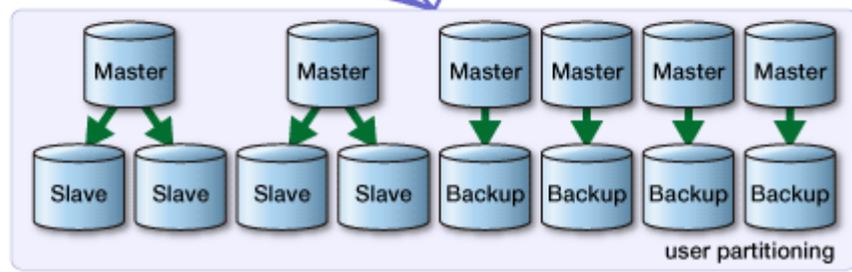
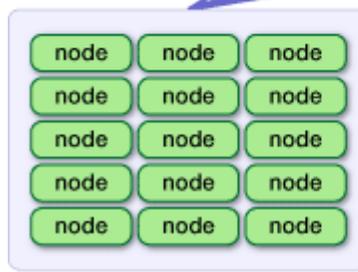
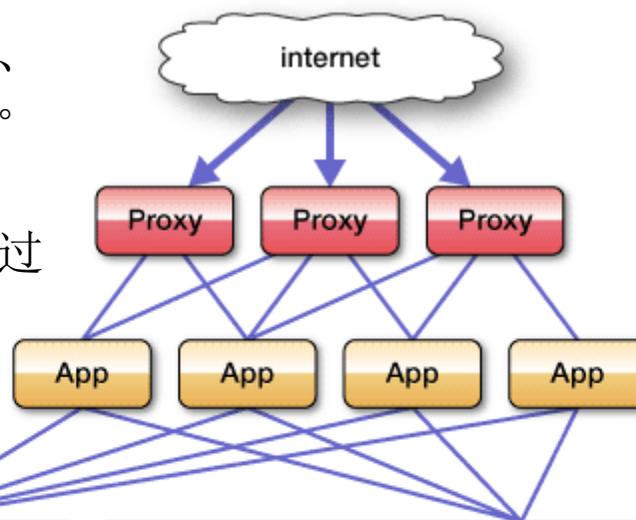
CPU访问L1就像从书桌拿一本书，L2是从书架拿一本书，L3是从客厅桌子上拿一本书，访问主存就像骑车去社区图书馆拿一书



MySQL MemCached

MemCached:

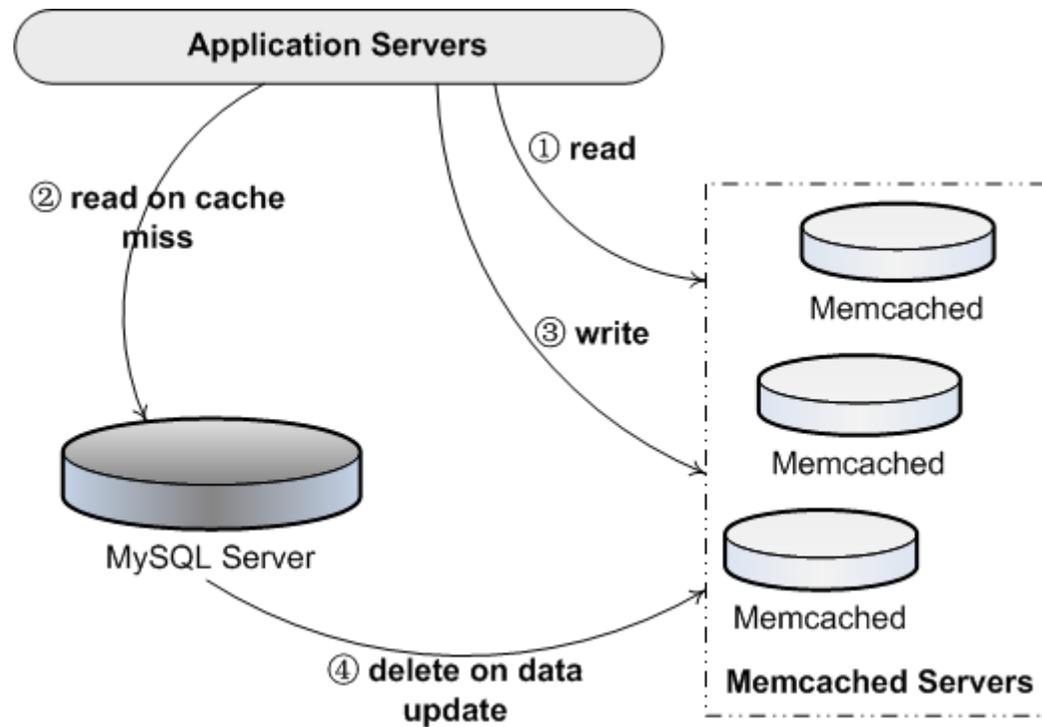
- 一个高性能的分布式内存对象缓存系统，用于动态Web应用以减轻DB负载；
- 它通过在内存中缓存数据和对象来减少读取DB的次数，从而提供动态、DB驱动网站的速度；
- 用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。
- 基于一个存储键/值对的HashMap；
- 通过LRU算法进行淘汰，同时可以通过删除和设置失效时间来淘汰存放在内存的数据
- 一致性hash解决增加或减少缓存服务器导致重新hash带来的大量缓存失效的弊端。

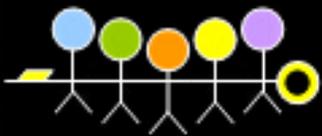


Ruby on Rails' Cache Money (twitter)



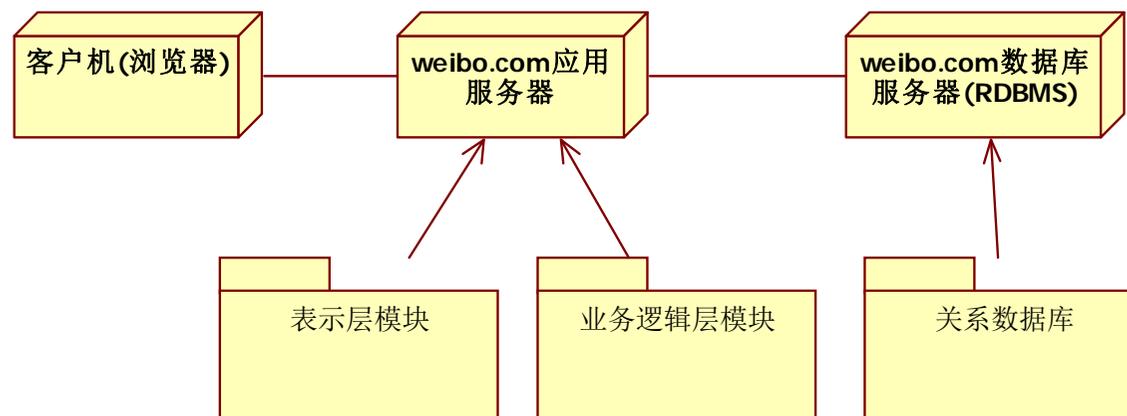
MySQL MemCached





微博的架构设计

- 最初：最简单的三层B/S架构
 - 所有程序均部署于应用服务器之上；
 - 专门的DB Server；

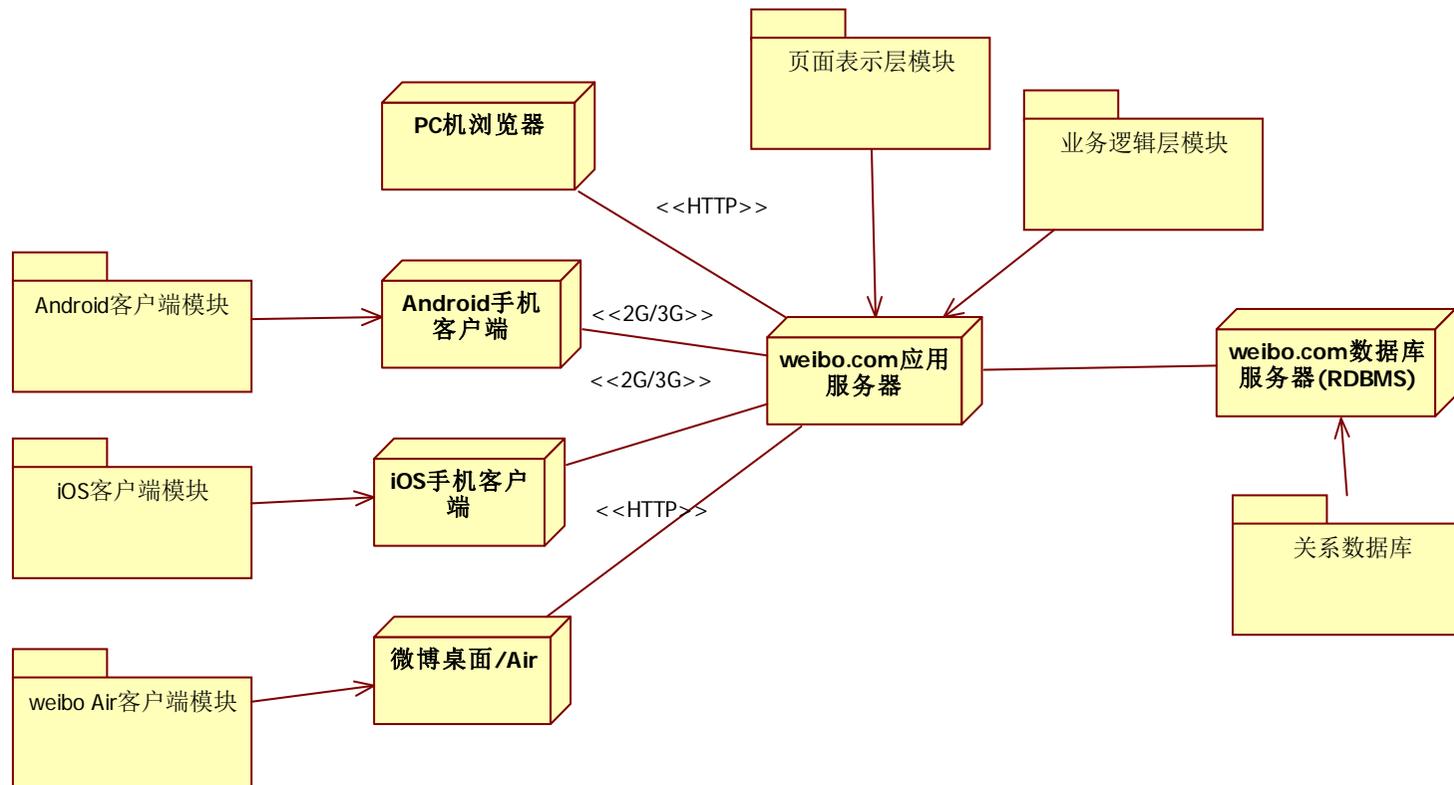


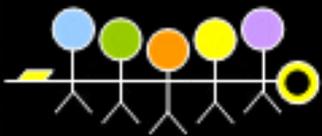


微博的架构设计

■ 考虑客户端的跨设备性NFR:

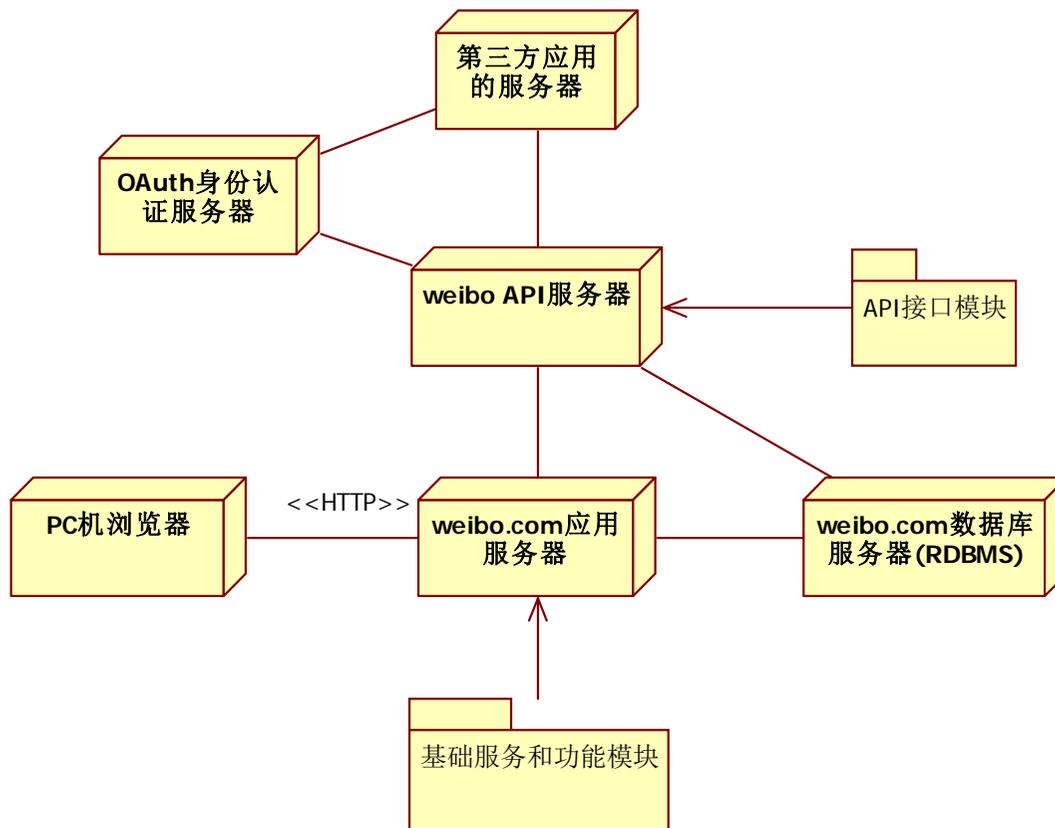
- 决策1: 针对每种不同类型的终端, 分别开发相应的客户端程序
- 决策2: 针对各种终端, 开发统一的基于浏览器的程序(例如HTML5)

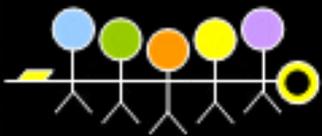




微博的架构设计

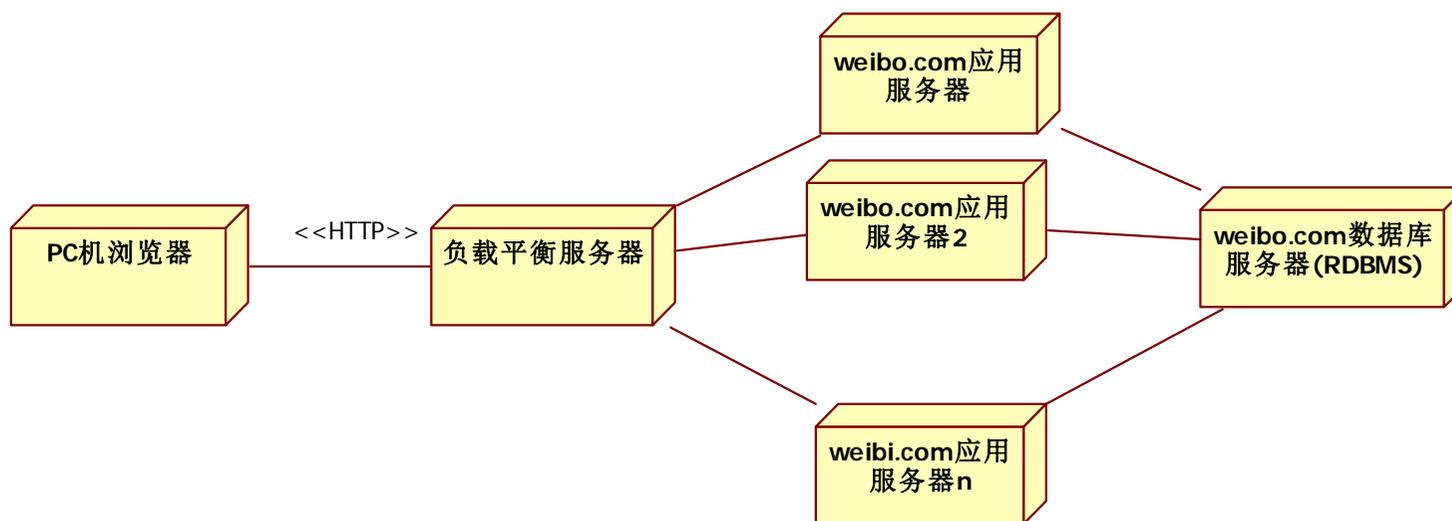
- 考虑开放性：
 - 决策：逻辑分层，内部功能封装，开放接口，统一身份认证， ...





微博的架构设计

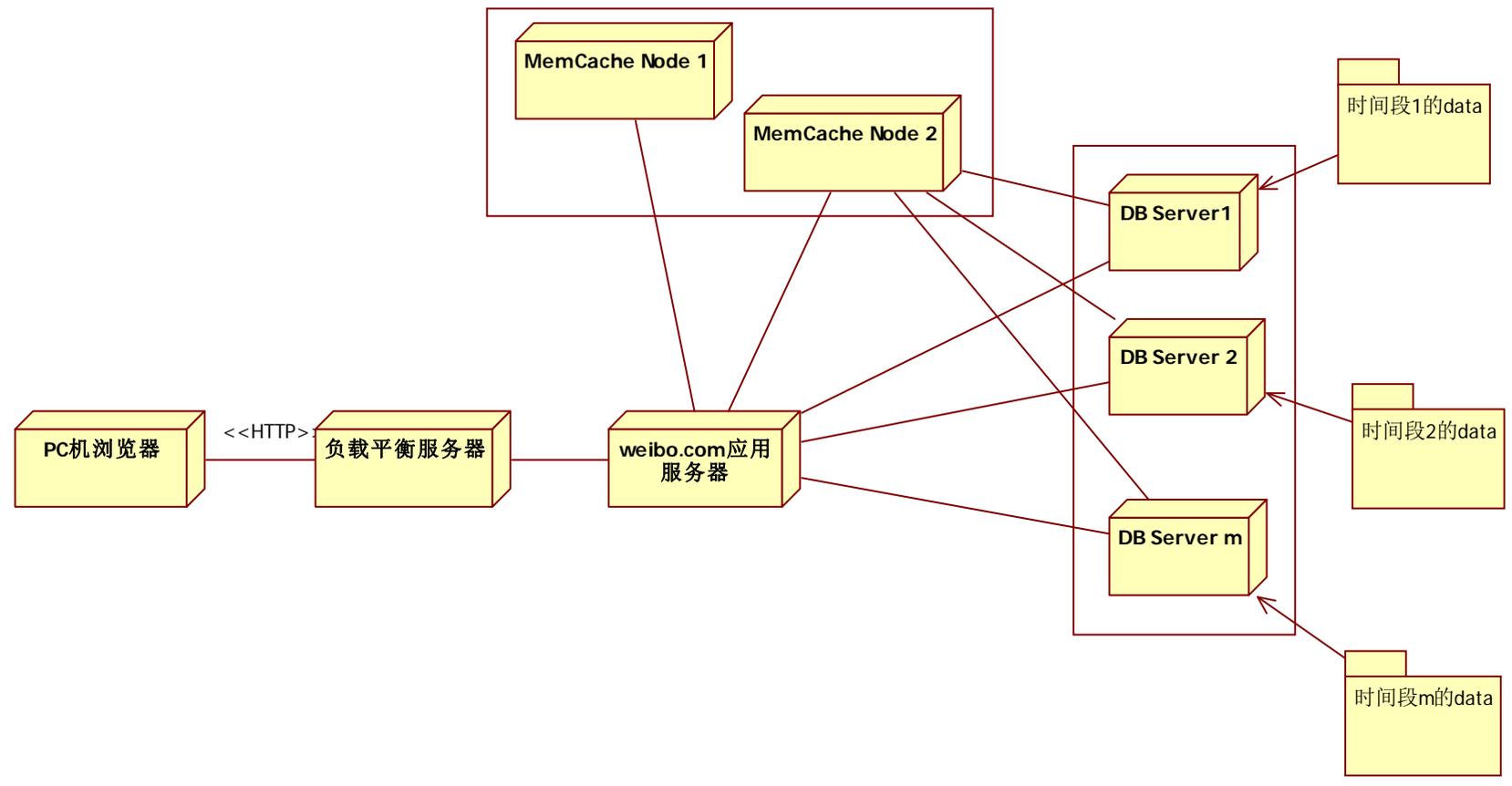
- 考虑吞吐率、伸缩性和可靠性：
 - 决策：采用集群策略





微博的架构设计

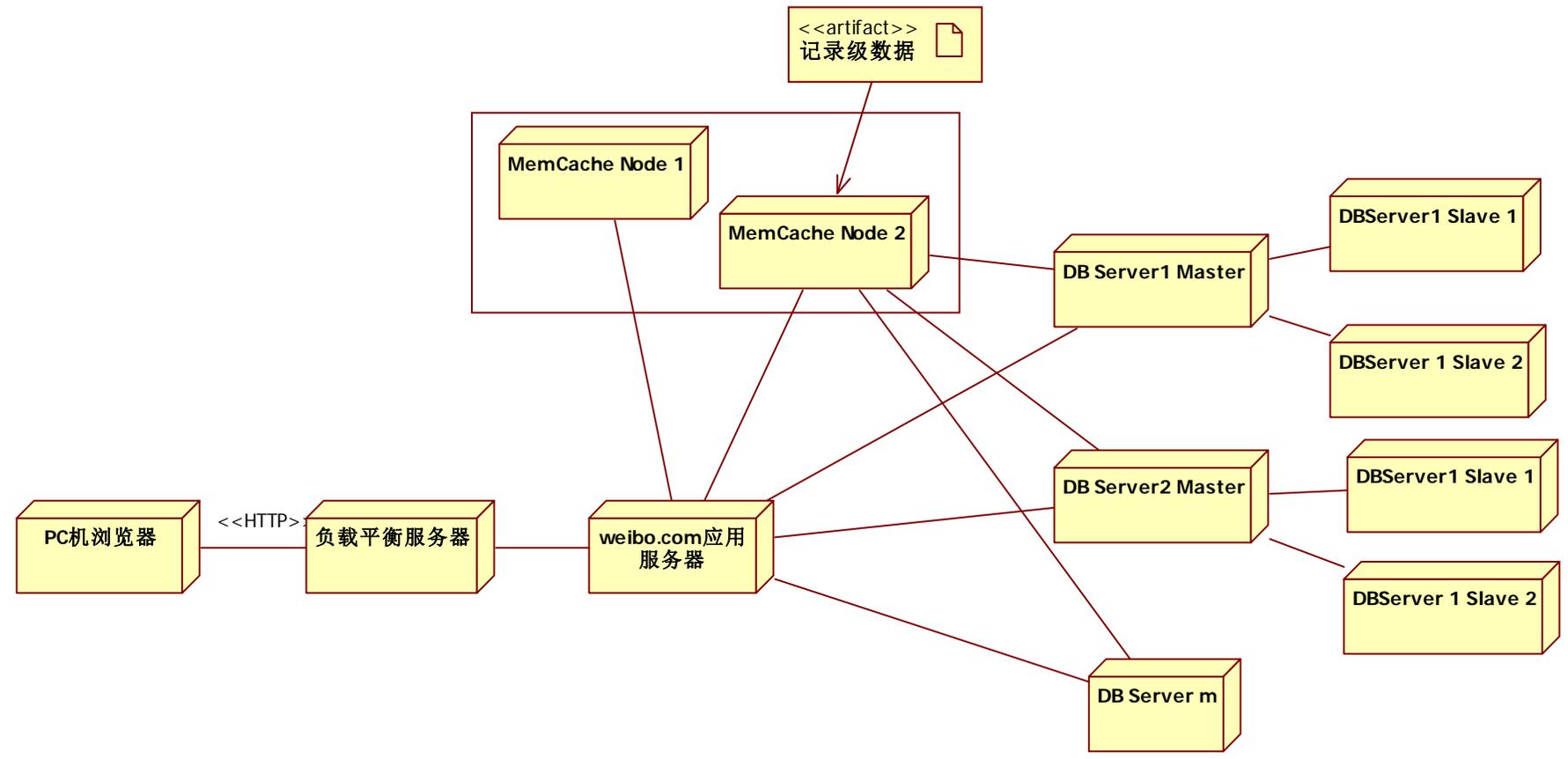
- 考虑吞吐率、数据存取效率、伸缩性：
 - 决策：采用数据分片(分布式数据库)、MemCache缓存机制

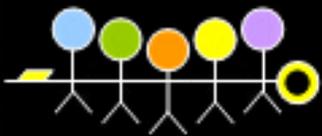




微博的架构设计

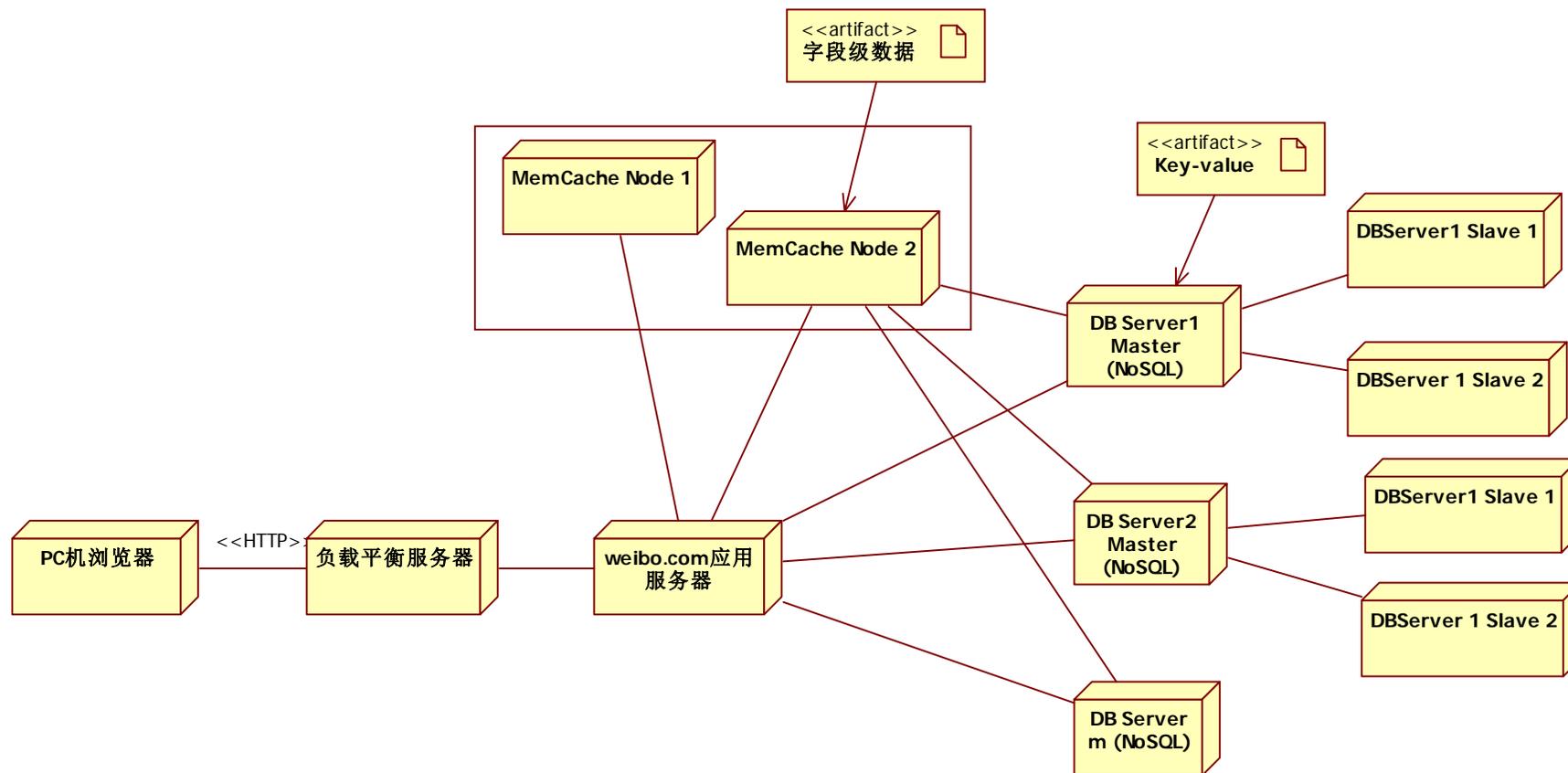
- 考虑可靠性：
 - 决策：数据库采用主从机制





微博的架构设计

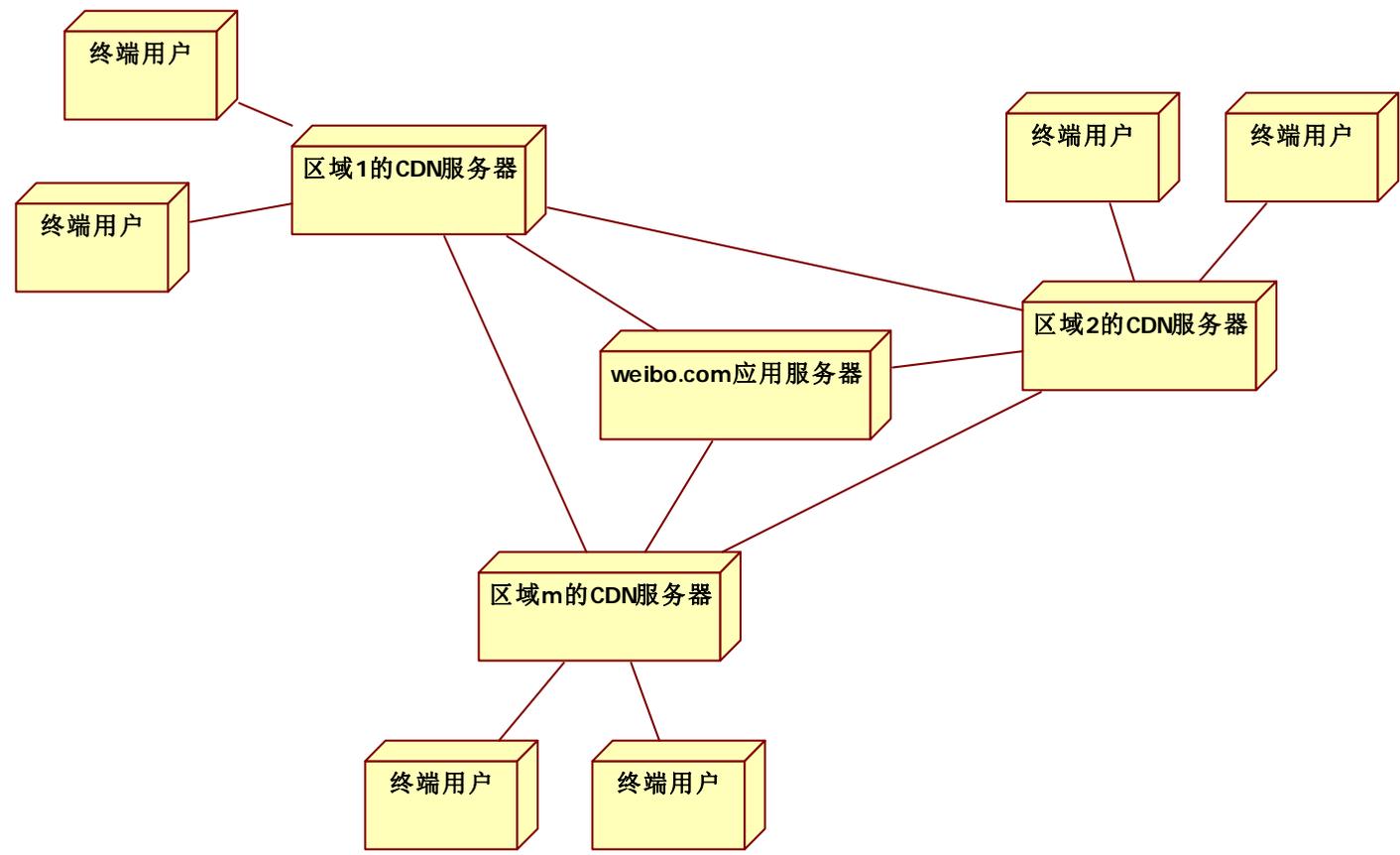
- 考虑伸缩性、可靠性、吞吐率：
 - 决策：NoSQL —— 采用Key-Value的方式存储/cache数据，数据的粒度更小，数据格式可灵活变化，etc

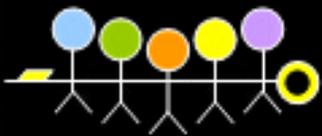




微博的架构设计

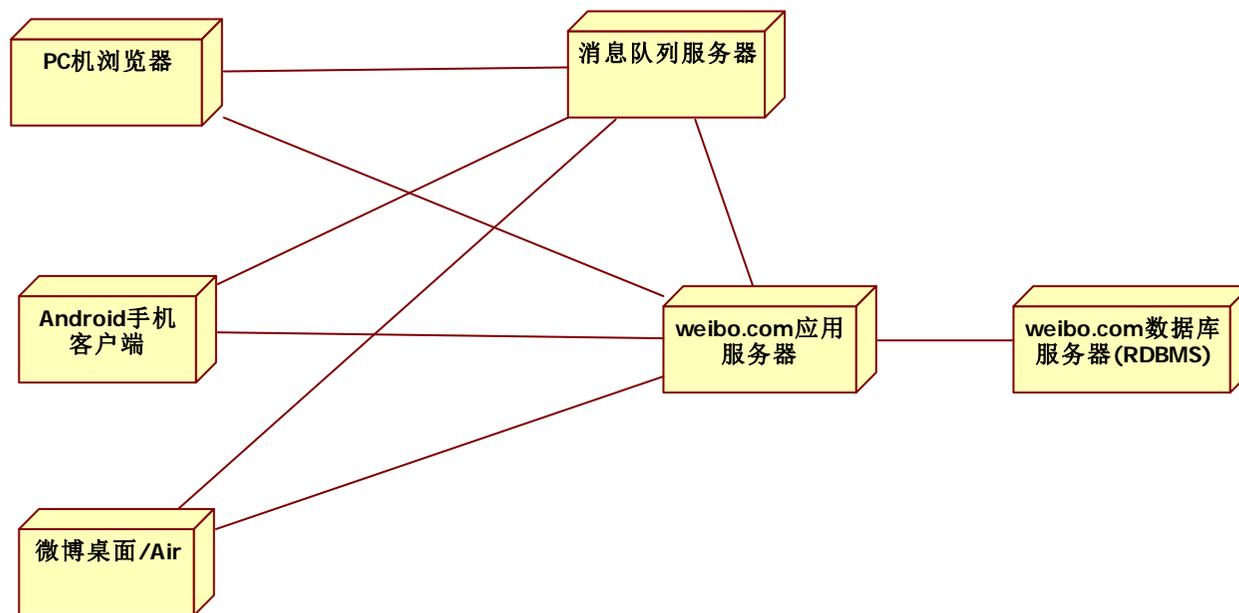
- 考虑吞吐率：
 - 决策：CDN

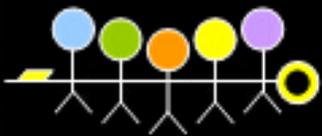




微博的架构设计

- 考虑吞吐率：
 - 决策：异步消息推送

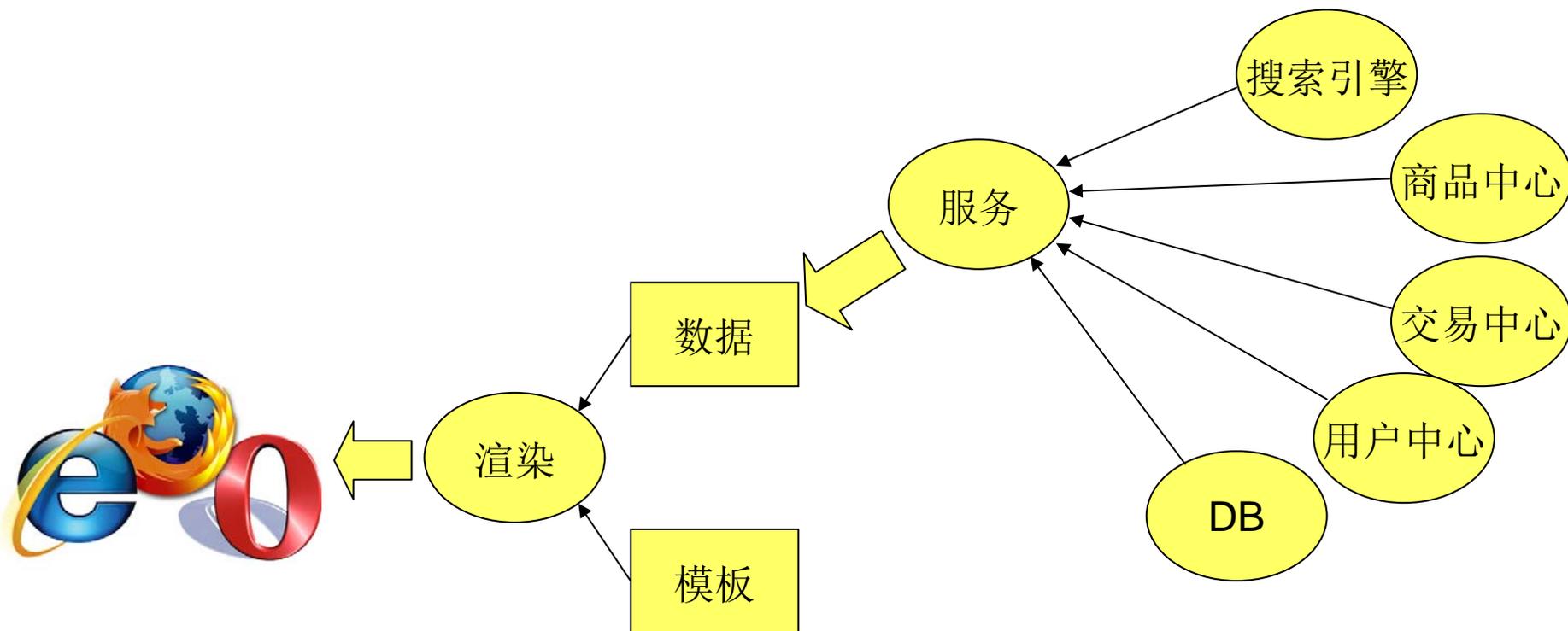




微博的架构设计

■ 考虑性能:

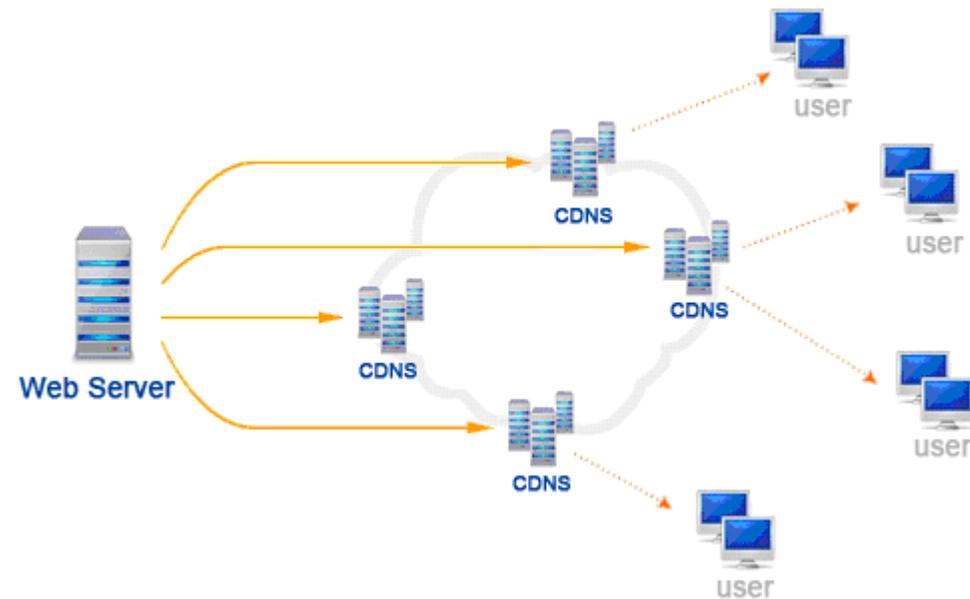
- 决策: 碎片化、异步渲染; 静态部分和动态部分分离, 分别提供——利用专用服务器的优势;
- 用户的一次请求被分解为多个小请求, 分别发往不同的服务器执行, 客户端通过异步加载+cache的方式将结果拼接起来。

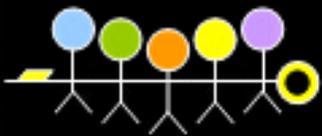




CDN

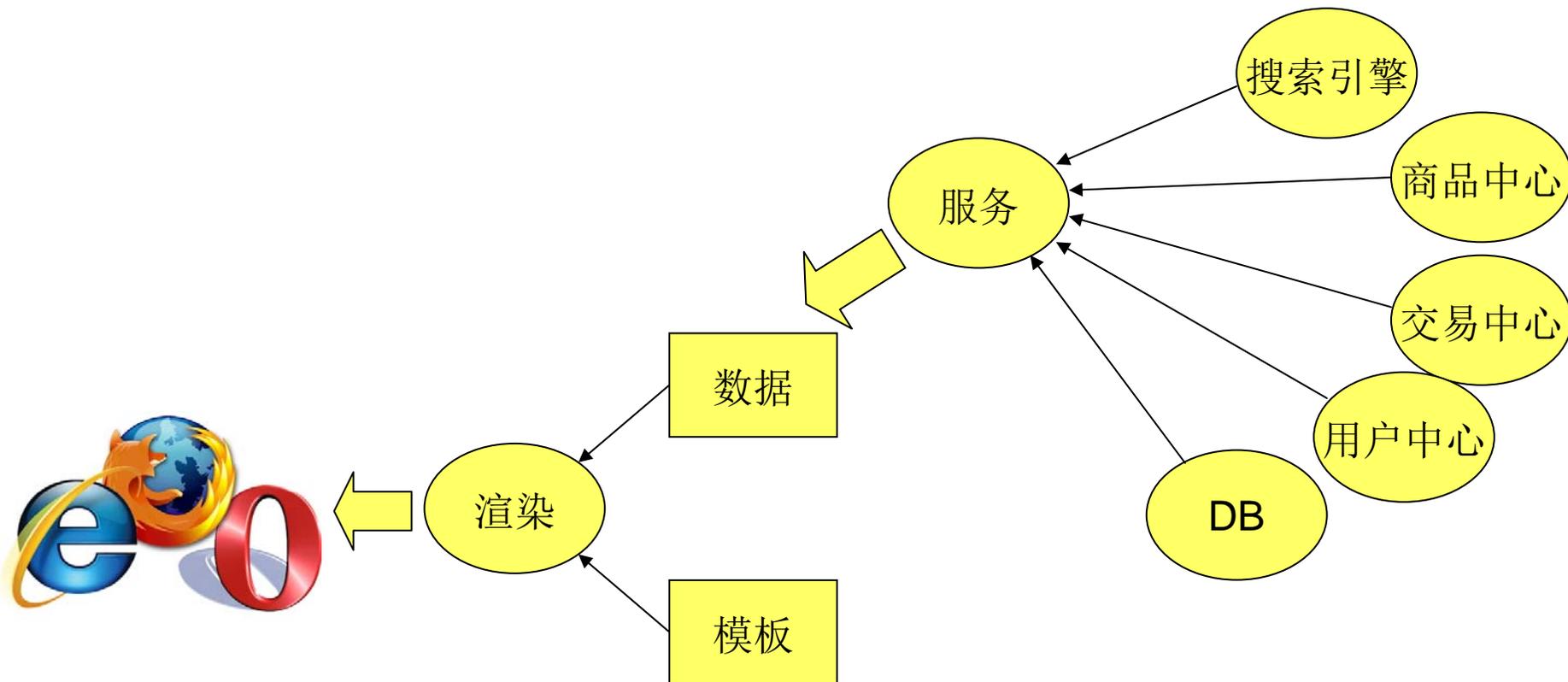
- **CDN(Content Delivery Network): 内容分发网络**
 - 通过在现有的Internet中增加一层新的网络架构，将网站的内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容。
 - 解决了由于网络带宽小、用户访问量大、网点分布不均等原因所造成的用户访问网站响应速度慢的问题。
- **CDN: 一个服务器的内容，平均分部到多个服务器上，服务器智能识别，让用户获取离用户最近的服务器，提高速度。**
 - 分布式存储
 - 负载均衡
 - 网络请求的重定向
 - 内容管理

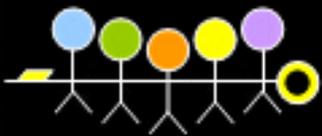




服务碎片化

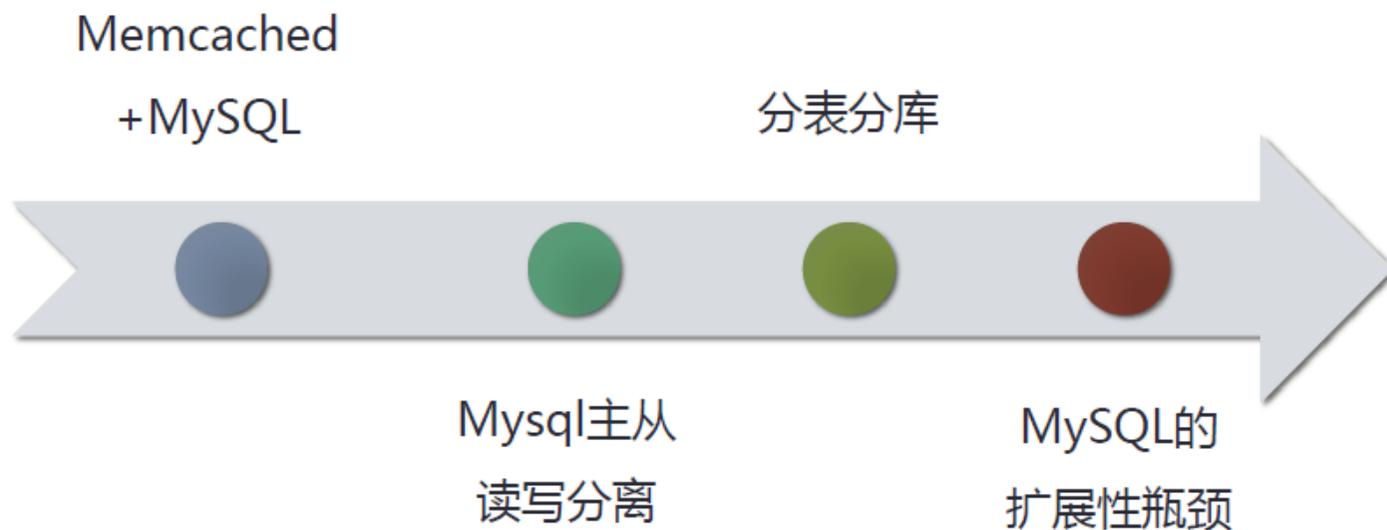
- 异步渲染
 - 静态部分和动态部分分离，分别提供——利用专用服务器的优势；
- 本质上：用户的一次请求被分解为多个小请求，分别发往不同的服务器执行，客户端通过异步加载+cache的方式将结果拼接起来。





服务数据

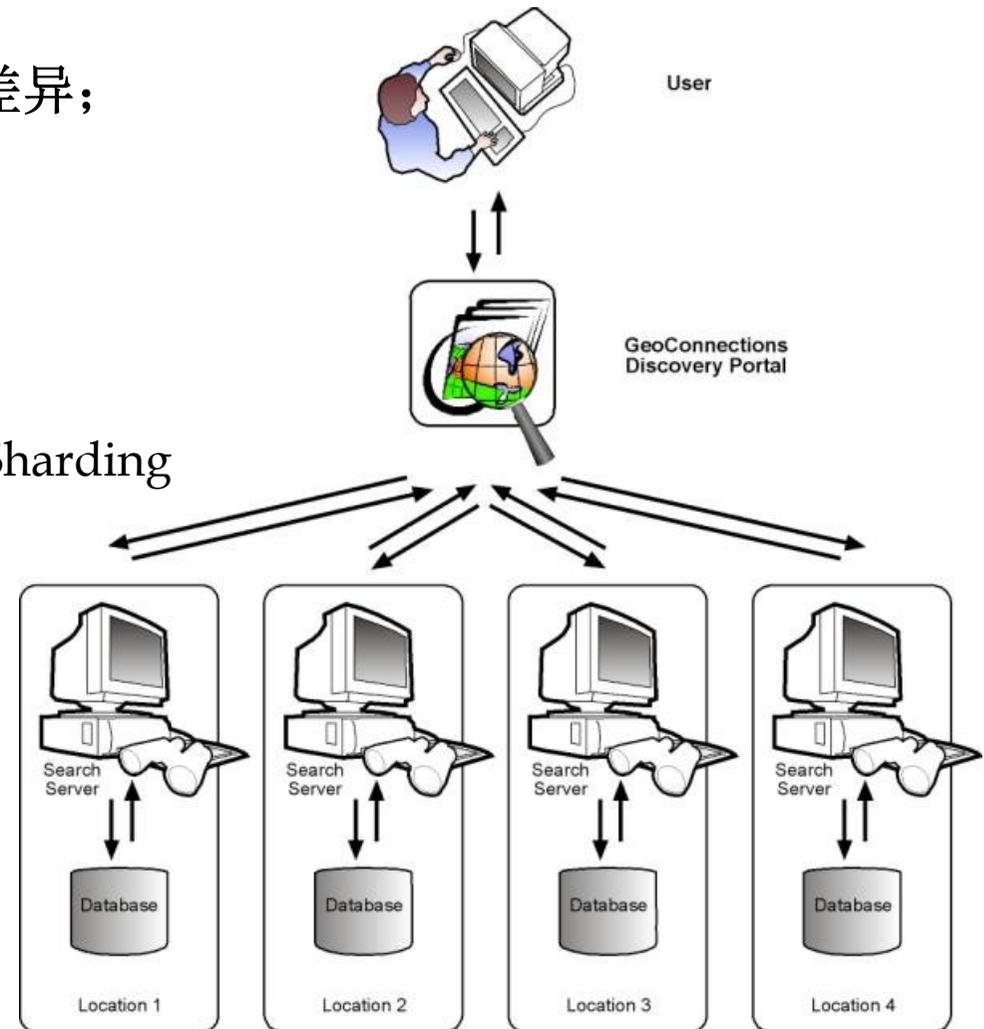
- 数据采集、数据存储、数据挖掘

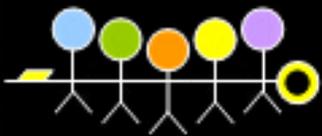




海量数据存储与检索

- 数据海量;
 - 访问数据的网络速度和带宽存在差异;
 - →需要数据的分布式存储;
-
- 解决方案:
 - 分布式数据库(Distributed-DB) & Sharding
 - Master/Slave
 - Master/Master
 - 2PC/3PC
 - Paxos

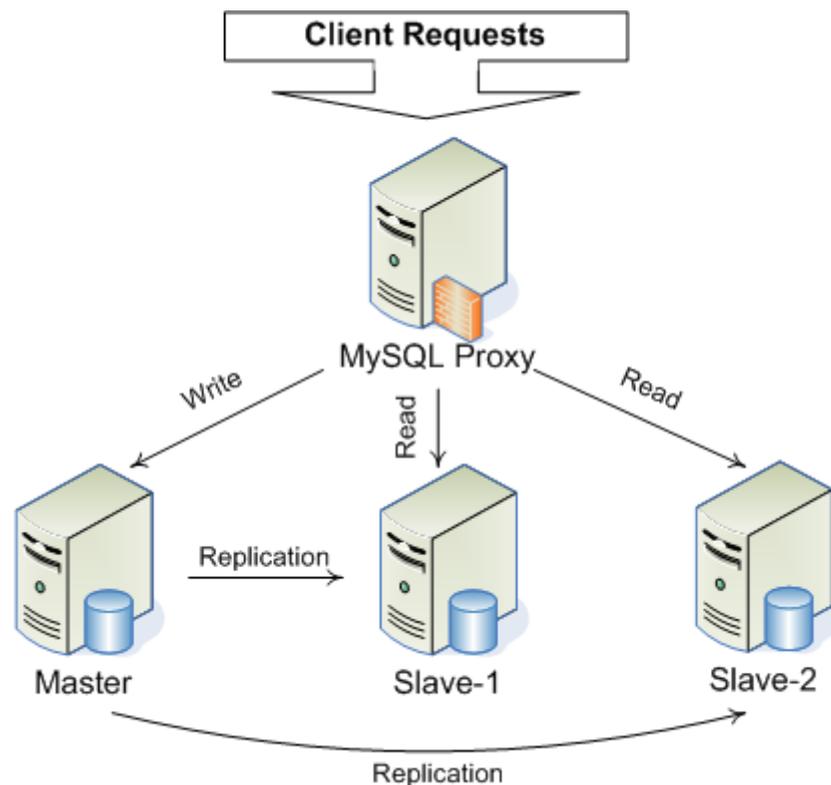




MySQL Master/Slave, Proxy

■ MySQL Master/Slave, Proxy

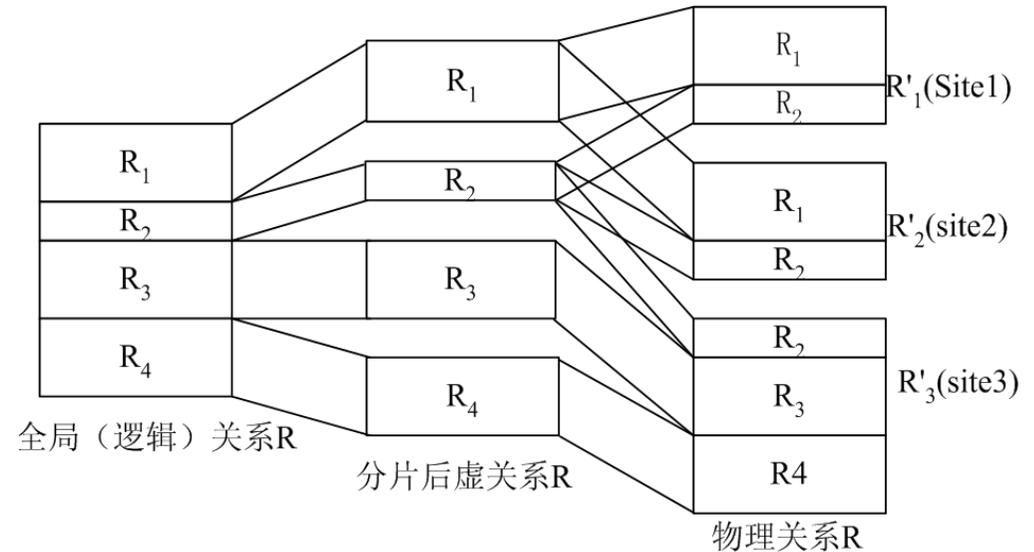
- 数据同步，故障转移处理，灾备 → 确保可靠性；
- 读写分离(read/write splitting)，实现负载均衡
- 让主数据库处理事务性查询，而从数据库处理SELECT查询。
- 数据库复制被用来把事务性查询导致的变更同步到集群中的从数据库。

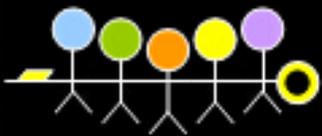




数据分片 (Data Sharding)

- **Sharding:** 对数据的分片
- 对数据进行分片的原因:
 - 随着SNS/eBusiness的数据规模不断膨胀，单个DB无法满足性能和存储空间要求；
- 分片可以达到的效果:
 - 可扩展性(scale out, 水平扩展)；
 - 对数据访问I/O的负载平衡；
- **数据分片的策略:**
 - 按用户UID切分——根据用户号码段范围进行切片，把不同的群体划分到不同的DB上
 - 按时间切分——交易中的数据与交易完成的数据划分到不同DB上；
 - 按访问热点切分——10%的“热”数据和90%的“冷”数据划分开来；





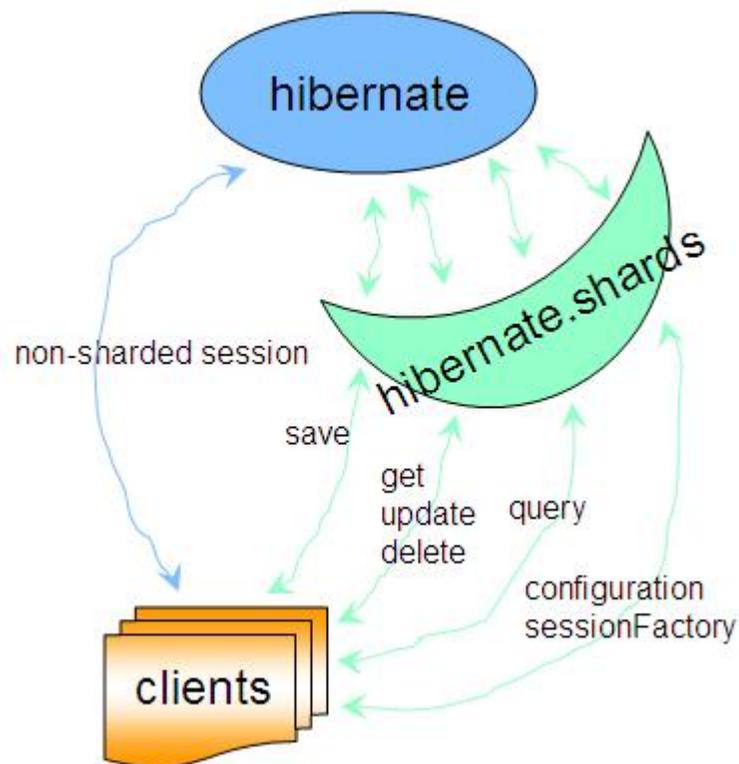
数据分片 (Data Sharding)

■ 新浪微博的数据分片策略:

- 按照时间拆分: 一个月的数据存储于单独的一张表;
- 内容和索引分开存放: “微博地址”是索引数据, “微博内容”是内容数据,
- 二次索引: 为保证按时间分片之后的检索方便, 把每个月记录的偏移记录下来, 以快速查询到用户微博数据。

■ Sharding的相关软件:

- MySQL Proxy + HSCALE
- Hibernate Shards (by Google)
- Spock Proxy
- HiveDB
- 等





NoSQL的兴起

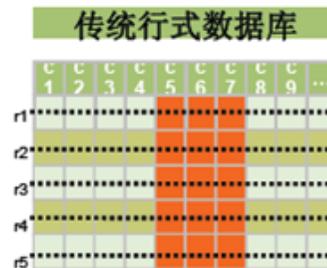
- 对数据管理的新需求：
 - High performance - 对数据库高并发读写的需求
 - 数据库并发负载非常高，往往要达到每秒上万次读写请求，硬盘IO无法承受；
 - Huge Storage - 对海量数据的高效率存储和访问的需求
 - 海量数据进行SQL查询，效率是极其低下乃至不可忍受的；
 - High Scalability && High Availability- 对数据库的高可扩展性和高可用性的需求
 - 无法通过简单的通过添加更多的硬件和服务节点来扩展性能和负载能力，需要停机维护和数据迁移
- **ACID**变得无用武之地
 - 并不要求严格的数据库事务，对读写一致性的要求很低；
 - 复杂的SQL查询(特别是多表关联查询的需求)越来越少；
- **NoSQL**: 打破关系数据模式，不需要固定的表结构，通常也不存在连接操作



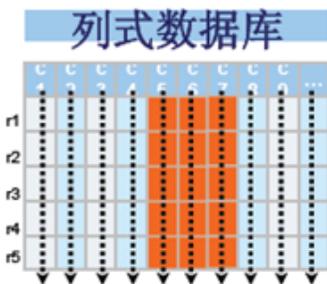
NoSQL



- NoSQL == Not Only SQL (不仅仅是SQL、非关系型数据存储)
- 不需要固定的表结构，也不存在连接操作，在大数据存取上具备RDBMS无法比拟的性能优势。
 - 假设失效是必然发生的
 - 对数据进行分片
 - 保存同一数据的多个副本
 - 动态扩展
 - 基于磁盘的和内存中的实现
 - 基于Key-Value的数据存储：
可以在系统运行时随意添加或删除字段



- 数据是按行存储的
- 没有索引的查询使用大量I/O
- 建立索引和物化视图需要花费大量时间和资源
- 面对查询的需求，数据库必须被大量膨胀才能满足性能要求



- 数据按列存储 - 每一列单独存放
- 数据即是索引
- 只访问查询涉及的列 - 大量降低系统I/O
- 每一列由一个线索来处理 - 查询的并发处理
- 数据类型一致，数据特征相似 - 高效压缩



结束

2014年11月10日

